Aprenda JavaScript com Dashboards

Seus primeiros passos na programação com grandes resultados

Casa do Código | alura

LUCAS TAUIL

Sumário

- ISBN
- O que você encontrará aqui
- Parte I — Entendendo as ferramentas
  - 1 O caminho até os Dashboards
  - 2 Estrutura e os primeiros códigos
- Parte II — Explorando o poder do JavaScript
  - 3 Bem-vindo ao JavaScript
  - 4 Funções e estruturas de repetição
  - 5 Trabalhando com dados em JavaScript
  - 6 Abstrações e lógicas avançadas
  - 7 JavaScript na Internet
- Parte III — Chegou a hora dos Dashboards!
  - 8 Visualização e preparação dos dados
  - 9 Gerando os Dashboards
  - 10 Importador inteligente de dados
  - 11 Visualizando o Brasil: Dados em mapa

ISBN

Impresso: 978-85-5519-405-4

Digital: 978-85-5519-406-1

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Tauil, Lucas
Aprenda JavaScript com dashboards : seus primeiros passos na programação com grandes resultados / Lucas Tauil. -- São Paulo : Casa do Código Livros e Tecnologia, 2025.

ISBN 978-85-5519-405-4

1. Ciência da computação - Estudo e ensino
2. JavaScript (Linguagem de programação)
3. Programação (Computadores) - Estudo e ensino
4. Tecnologia I. Título.

25-294949.0 CDD-005.133

Índices para catálogo sistemático:

1. Java : Linguagem de programação : Computadores : Processamento de dados 005.133

Livia Dias Vaz - Bibliotecária - CRB-8/9638

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

O que você encontrará aqui

Este livro é um guia elaborado de forma didática para você que deseja começar no mundo da programação. Apresentaremos, desde o básico, a linguagem web mais popular do mundo: o JavaScript. Todo o desenvolvimento será direcionado à criação de painéis informativos, os famosos Dashboards.

Essa abordagem busca atrair pessoas que já possuem uma massa de dados, mas enfrentam dificuldades para transformá-la em informação — seja por falta de familiaridade com ferramentas técnicas ou pelo alto custo de softwares especializados. Tudo o que usaremos aqui serão ferramentas gratuitas.

Você deve estar se perguntando: "Mas meus dados estão em diferentes formatos, como eles serão colocados em um Dashboard?". Aprenderemos a trabalhar com dados a partir do formato *.csv*, que é basicamente um formato adaptado de qualquer planilha de dados. Embora seja útil ter dados em formato de planilha, não é obrigatório para acompanhar o conteúdo deste livro.

Este livro vai ensinar você a programar em JavaScript, desde o comando mais simples, como um *alert("Hello World")*, até a criação de Dashboards atraentes e capazes de serem visualizados em qualquer dispositivo, seja móvel ou não (celulares ou computadores).

Compartilho aqui um compilado dos meus 12 anos de experiência como desenvolvedor de aplicações para web e mais 4 na docência na área de tecnologia, para mostrar que, com as ferramentas mais básicas e gratuitas da tecnologia, é possível criar aplicativos muito robustos.

Essas informações ajudarão você a conhecer a programação e vão fornecer resultados rápidos e atraentes, quebrando o paradigma de que são necessários conhecimentos aprofundados para começar a trabalhar na área de TI. Além disso, este livro pode abrir as portas

do mundo da tecnologia e incentivar o avanço em uma das áreas com maior demanda de vagas de trabalho atualmente. Inicie já! Tudo o que você precisa já está no seu computador e, agora, em suas mãos.

Para quem é este livro?

- Iniciantes em lógica de programação;
- Iniciantes da área da tecnologia que desejam conhecer e praticar com a linguagem web mais popular do mundo;
- Pessoas interessadas em transformar dados brutos em informações visuais para tomada de decisão.

É necessário algum pré-requisito?

Não. Este livro apresentará como os sites e plataformas web são codificados e organizados nos seus pilares mais básicos. Ao final deste livro, você será capaz de compreender o funcionamento básico existente por trás de toda página da internet.

Qualquer site pode ser composto por diversas tecnologias, sejam básicas ou avançadas, mas todas derivadas do HTML, CSS e JavaScript, que são as linguagens mais trabalhadas aqui. Por isso, este livro é perfeito para iniciantes do mundo da programação.

Sobre o livro

Este livro foi escrito pensando em você que está começando sua jornada no mundo da programação e quer ver resultados práticos desde os primeiros passos. A proposta é mostrar que, mesmo com conhecimentos iniciais, é possível criar aplicações web úteis, modernas e, principalmente, visuais.

Para isso, o foco será a construção de Dashboards interativos, usando apenas ferramentas gratuitas, acessíveis e que funcionam diretamente no navegador.

O livro foi dividido em três partes, com uma evolução pensada para que o aprendizado seja gradual e sem saltos conceituais.

Parte I — Entendendo as ferramentas

Aqui, você conhecerá os principais conceitos por trás de qualquer página web. Vamos falar sobre editores de código, navegadores, e os três pilares da web: HTML, CSS e JavaScript. Tudo apresentado de forma simples e com exemplos práticos para garantir que ninguém fique perdido.

Parte II — Explorando o poder do JavaScript

Depois de dominar o básico da web, você entrará no universo da programação com JavaScript. Nesta parte, vamos abordar desde conceitos fundamentais como variáveis, funções e estruturas de repetição até temas mais avançados como manipulação de objetos, *arrays*, eventos e consumo de APIs. A ideia é que você ganhe fluência suficiente na linguagem para criar interações reais com os usuários.

Parte III — Chegou a hora dos Dashboards!

Com a base construída, chega o momento mais esperado: a geração dos Dashboards. Você aprenderá a preparar os dados, criar Dashboards, mapas interativos e visualizações que realmente transformam informações brutas em *insights* visuais. Cada capítulo desta parte traz um exemplo prático, incluindo Dashboards de linha, comparações temporais, análises geográficas e até um importador de dados.

Além de aprender a codificar, você também terá contato com boas práticas de organização, estruturação de código e formas de deixar seus projetos web mais profissionais.

O livro mantém uma linguagem acessível e direta, sem exigir conhecimentos prévios em programação. Tudo é ensinado do zero, mas com a profundidade necessária para que, ao final, você se sinta capaz de criar seus próprios projetos. Mais do que ensinar códigos, a proposta aqui é mostrar que você também pode contar histórias com dados. E que a tecnologia pode ser uma ponte entre informação e entendimento.

Sobre o autor

Lucas Chasseraux Tauil é desenvolvedor web desde 2010. Passou por várias tecnologias em sua carreira e em 2022 se tornou professor universitário ministrando disciplinas relacionadas a raciocínio lógico, desenvolvimento de software, redes de computadores e sistemas operacionais.

É graduado em Ciência da Computação, pós-graduado em Redes de Computadores e em Desenvolvimento Web. Atualmente, cursa mestrado em Ciência da Computação aprofundando ainda mais seus estudos na área.

Pode ser encontrado em seu LinkedIn https://www.linkedin.com/in/lucastauil/ e em sua página pessoal https://espaco.dev.

Agradecimentos

Este livro é dedicado à minha esposa, a Pri, que acredita em mim antes mesmo de eu acreditar. Ela me incentiva a sair da zona de conforto e, mesmo que não entenda algumas das minhas decisões mais incomuns, sempre apoiou minhas teimosias profissionais e nunca me deixou desistir — ainda que isso nos custasse adaptações na rotina ou no aspecto financeiro. Ter uma companheira como ela é certamente algo de outras vidas...

Parte I — Entendendo as ferramentas

Nesta primeira parte do livro, vamos entender as ferramentas básicas para construção de Dashboards interativos.

Você verá:

- Como se organiza o processo de criação de Dashboards na web.
- O papel de editores de código e navegadores.
- Fundamentos essenciais de HTML, CSS e JavaScript.
- O que são e quais as bibliotecas usadas ao longo do livro, como o Chart.js.

Essa base vai preparar você para colocar a mão na massa com mais segurança nas partes seguintes.

Capítulo 1
O caminho até os Dashboards

Vamos entender os primeiros passos da nossa jornada. Isso envolve compreender nosso ponto de partida e o ponto de chegada. Como autor, apaixonado pela área, tenho acompanhado de forma significativa a expansão da Tecnologia da Informação (a famosa T.I.) em suas diversas ramificações. A pandemia da COVID-19 impulsionou a transformação digital de uma forma sem precedentes. Somente no Brasil, aproximadamente 46 mil profissionais são formados anualmente na área de tecnologia, mas a demanda gira em torno de 70 mil. Tenho percebido esse aumento não apenas em dados divulgados, mas também como docente. A procura é tão intensa que também puxa a demanda dos agentes multiplicadores: os professores.

Não quero me alongar nessas informações, porque meu intuito é fazer um livro mão na massa, mas quero enfatizar que nosso livro

não apenas ensinará você a montar Dashboards na web para analisar dados, ele vai abrir a porta de entrada para a área da tecnologia, proporcionando uma nova habilidade ou até mesmo capacitando você a se tornar uma pessoa desenvolvedora web. A ideia é democratizar a educação tecnológica.

Qualquer página da internet que você utiliza, seja em sites ou sistemas do seu local de trabalho, usa três tecnologias que são os pilares da web: HTML, CSS e JavaScript. Vamos abordar essas três linguagens de modo que possamos compreendê-las, inserindo-as no momento certo conforme o processo de aprendizado.

1.1 Editores e navegadores

Para iniciar no mundo da programação, você precisa basicamente de um editor, que é qualquer programa que permita escrever textos simples, sem formatação. Por exemplo, o Bloco de Notas instalado no seu próprio Windows. Por que enfatizei o “texto simples”? Se pensarmos no Microsoft Word, ele também é um editor de texto, mas contém recursos voltados para escritores. No nosso caso, escreveremos códigos — então, qualquer recurso além do próprio texto digitado não é necessário.

Se você abrir o seu Bloco de Notas agora, digitar os códigos a seguir e salvar esse arquivo com o nome "index.html", terá criado o seu primeiro site.

```
<html>
<body>
Meu primeiro site
</body>
</html>
```

Ao salvar seu arquivo, é necessário um primeiro ponto de atenção. Por padrão, o Windows vai tentar salvar seu arquivo com a extensão

`.txt` . Conforme indicado, você deve alterar o tipo do arquivo para "Todos os arquivos (`*.*`)" e salvar com o nome `index.html` .

Pronto, você fez o seu primeiro arquivo de site. Se você acessar o local onde salvou seu arquivo, vai observar que ele estará com um ícone do seu navegador favorito. Ao clicar nele, abrirá uma página da *web*.

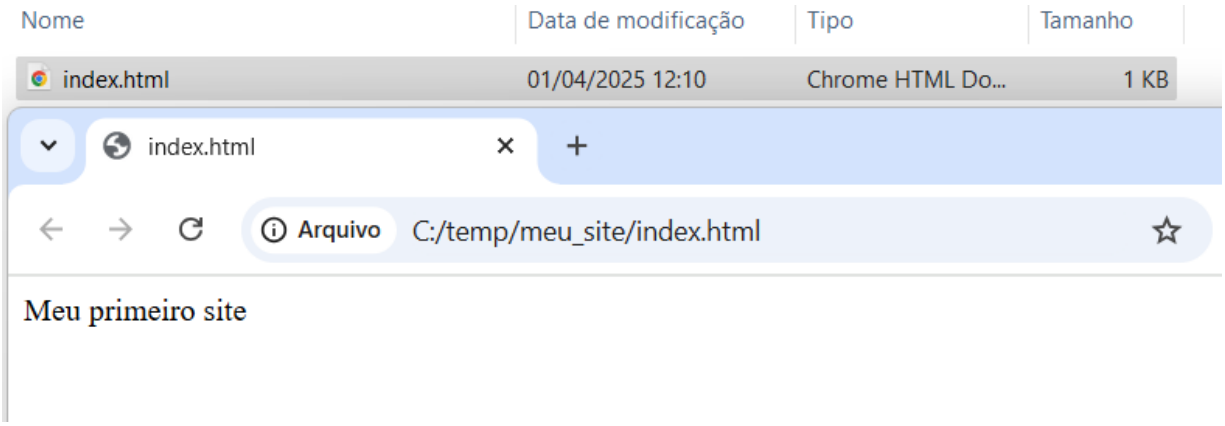


Figura 1.1: Visual do arquivo index.html na pasta e aberto no navegador

A premissa básica é essa. Codificar em um programa de texto e visualizar como está ficando nosso site.

Conforme nosso site vai sendo elaborado, o volume de código aumentará e, sejamos sinceros, seria horrível trabalhar com um grande código direto no Bloco de Notas. Por isso, para facilitar nossa vida, vou usar um programa gratuito que é um dos mais utilizados pelos programadores: o **Visual Studio Code**.

O uso deste programa não é obrigatório. Prometi para vocês no início deste livro que o seu computador já teria todas as ferramentas necessárias. Porém, por uma questão de facilidade, fica como sugestão este software gratuito. Se não desejar instalar nada, é só continuar com o Bloco de Notas. Mas vou mostrar a diferença entre a visualização do código no Bloco de Notas e no Visual Studio Code:

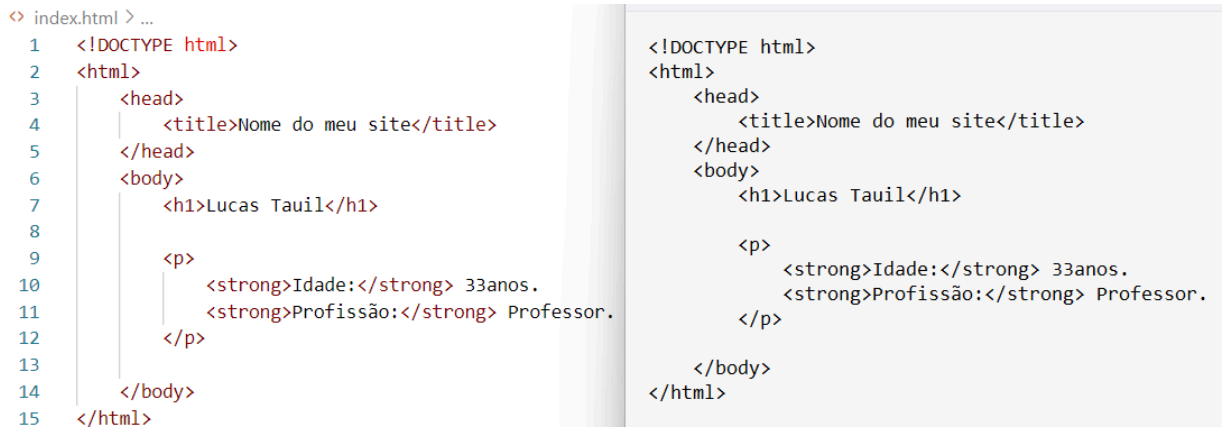


Figura 1.2: Highlights

À esquerda é possível ver um código no Visual Studio; à direita temos o código no Bloco de Notas. O resultado final desses dois códigos será o mesmo, mas observem que no Visual Studio Code existe realce de cores que distingue o código do texto comum. Agora, imaginem um código mais extenso. Com certeza o entendimento será mais simples com esses realces.

Assim, vamos utilizar o Visual Studio Code para facilitar o aprendizado. Se você também gostou dessa ferramenta e quiser instalar em seu computador, é só acessar o endereço eletrônico https://code.visualstudio.com/ e fazer o *download* dele. Fique tranquilo, é um programa leve e compatível com diversos sistemas operacionais, como Linux e Mac, caso o seu computador não utilize Windows.

Aqui é importante observar que, se você estiver lendo em um *e-reader* ou em uma edição impressa em preto e branco, pode ser que a diferença visual não seja tão evidente. Ainda assim, além da numeração das linhas e dos traços verticais que ajudam na organização do conteúdo, há também diferenciação por cores entre os elementos do código — o que será mais perceptível em telas coloridas.

A outra ferramenta de que vamos precisar é o navegador de sua preferência. O navegador será usado a cada passo do desenvolvimento para enxergarmos como está o andamento do nosso projeto. Afinal, é a partir dele que o produto final será visualizado. Neste livro usaremos o Google Chrome, mas sinta-se à vontade para escolher o navegador que desejar — todos eles apresentam o mesmo resultado e possuem uma ferramenta do desenvolvedor que vamos conhecer já.

Lembra-se do primeiro site que fizemos alguns parágrafos? Abra-o com o seu navegador e clique na tecla "F12". A maioria dos navegadores abrirá as "Ferramentas do desenvolvedor". Se o seu navegador não abrir, procure em "opções" ou "configurações" — lá você encontrará essa opção. Você verá algo parecido com isso:

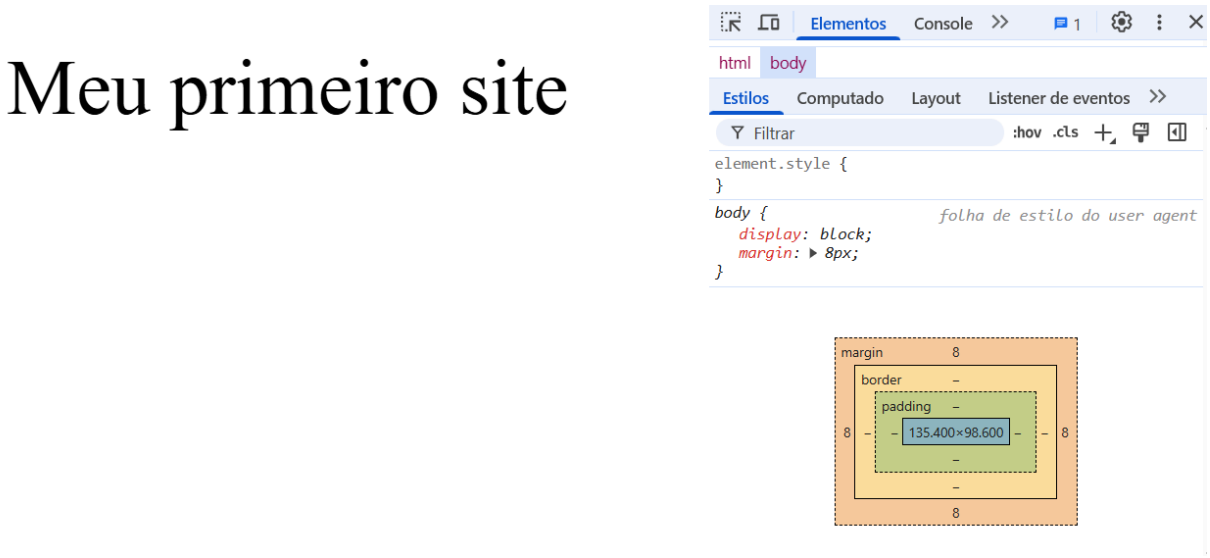


Figura 1.3: Ferramenta do Desenvolvedor

À esquerda é possível ver o seu primeiro site e à direita teremos as "Ferramentas do Desenvolvedor". Observe que essa ferramenta consegue enxergar o código que você escreveu no seu editor e ainda fornece algumas informações a mais, como esse quadrado que exibe propriedades como margens e bordas. Se você abrir essa ferramenta em qualquer site, ele vai mostrar um pouco do código

daquele site. Note ainda que essa ferramenta tem abas. Para o nosso JavaScript vamos utilizar muito a aba *Console*.

Vamos testá-la? Clique ali na aba *Console* e digite o comando:

```
alert('Bem vindo!');
```

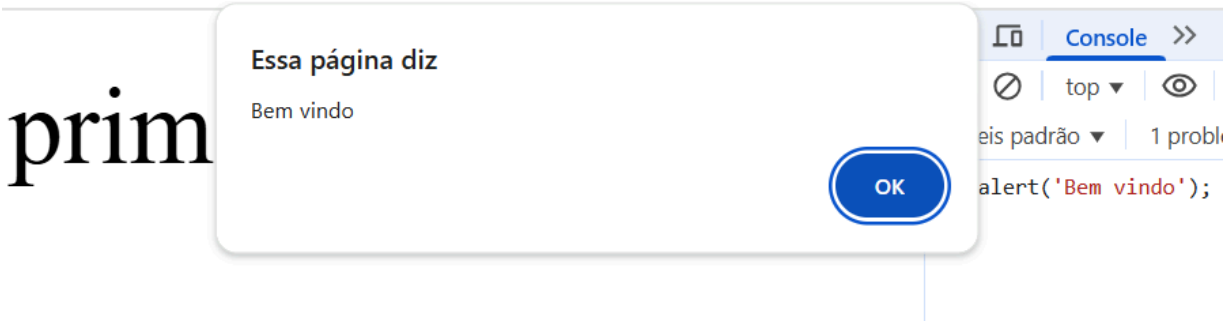


Figura 1.4: Exemplo do comando de alerta

Você acabou de executar um código JavaScript no seu navegador. Isso funciona só no meu site? Não. Se você abrir qualquer página e seguir os mesmos passos você terá:

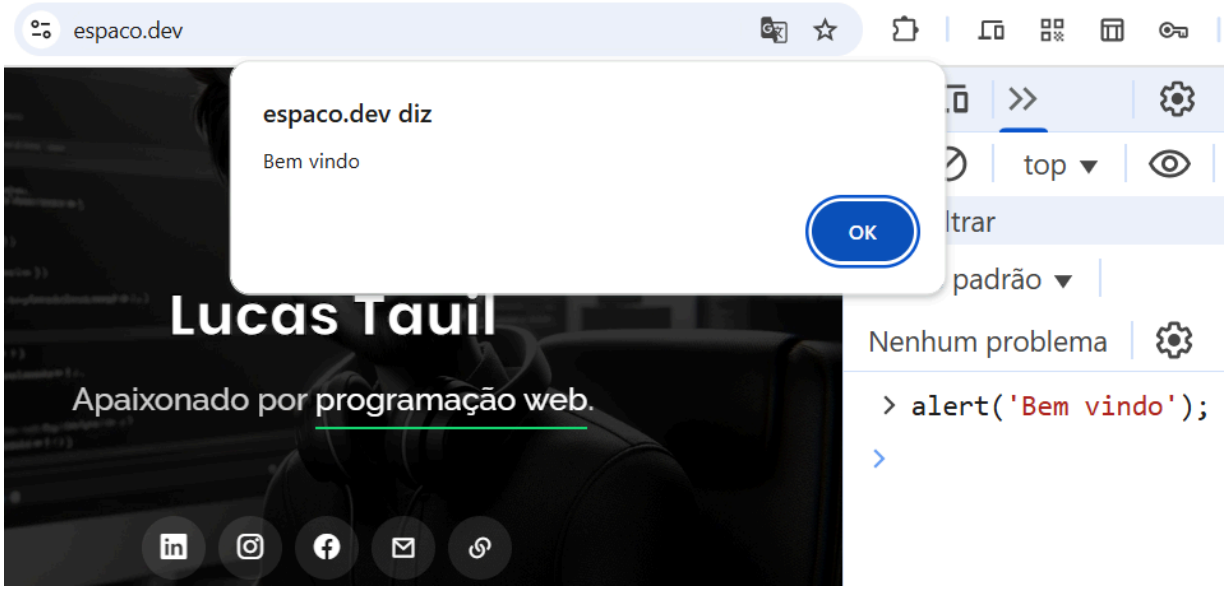


Figura 1.5: Exemplo do comando de alerta

Assim, concluímos nossas ferramentas de trabalho. Agora vamos conhecer as tecnologias que usaremos no processo.

1.2 Live Server

Já temos tudo que é necessário para programar, mas eu me sinto obrigado a citar os facilitadores. Tenha em mente que com um editor de códigos e o navegador escolhido você já tem todas as ferramentas para programar na web. Este tópico sobre o Live Server é a demonstração de um facilitador da jornada. Esse facilitador só estará disponível para quem optar em usar o Visual Studio Code como Editor de códigos.

Neste formato que vimos, você codifica seu programa, salva o arquivo e tem que ir lá na pasta e clicar no seu arquivo HTML para ver como ele está ficando. O Live Server é um *plugin* gratuito do Visual Studio Code que roda o seu código e, conforme você vai evoluindo, ele vai atualizando a visualização automaticamente para você.

Parece pequeno o auxílio deste plugin, mas esse ciclo de codificar, salvar, abrir o arquivo, voltar a codificar, salvar, abrir o arquivo, voltar a codificar, salvar, abrir o arquivo etc. se repete bastante quando estamos programando e um facilitador desse vai economizar muito tempo. Para instalá-lo no seu Visual Studio Code, você verá um menu lateral à esquerda chamado "Extensions":

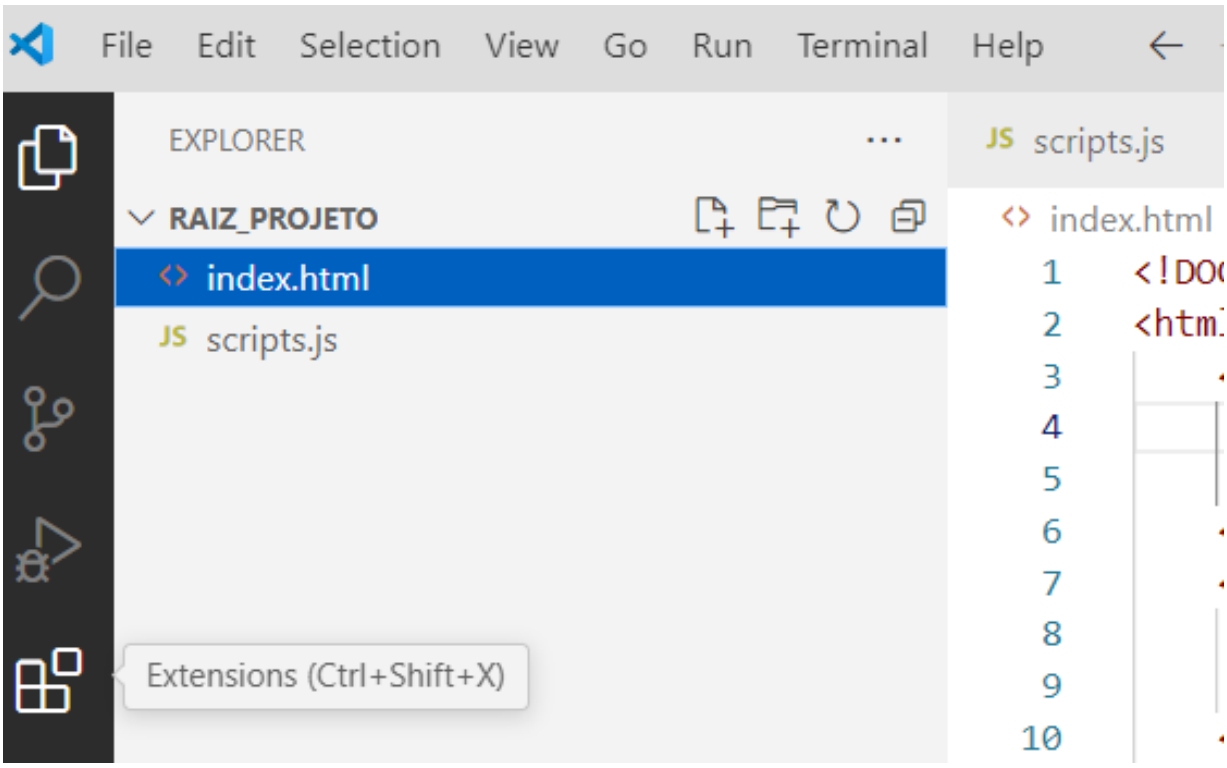


Figura 1.6: Ícone de extensões

Clique nessa opção e digite no campo de busca "Live Server" e vai aparecer para você alguns *plugins*. Escolha este conforme a imagem em que haverá um botão de instalar:

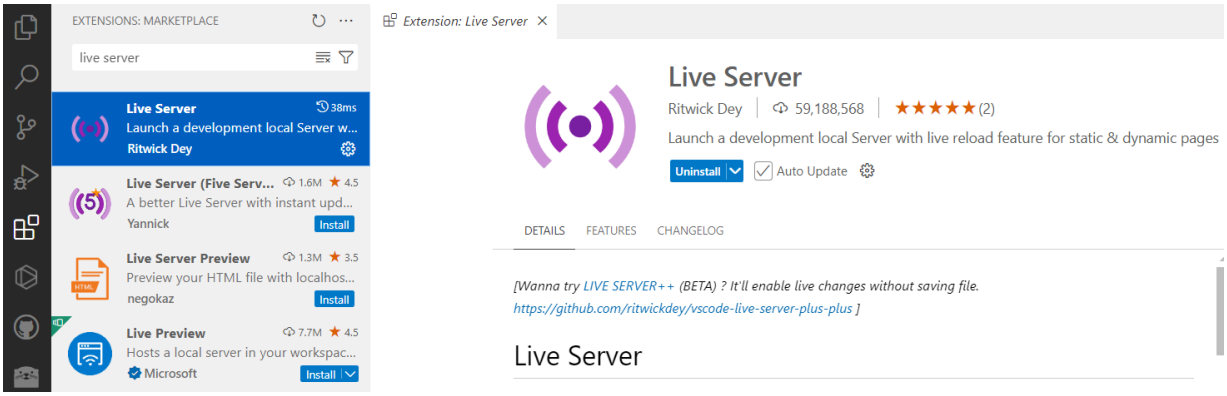


Figura 1.7: Instalando extensão Live Server

A instalação é bem rápida. Depois de instalar, vai aparecer no canto inferior esquerdo um ícone com uma antena escrito *Go Live* que, ao ser clicado, fará com que o código do seu projeto seja

automaticamente aberto no navegador. Deixe essa aba aberta e vá codificando. Conforme seu arquivo de código é salvo, essa aba do navegador se atualiza automaticamente.

1.3 Os pilares da Web

Diríamos que estamos com todo o ferramental pronto: os programas e *plugins* que vão nos ajudar a escrever códigos. Agora vamos falar um pouco das tecnologias que serão usadas: HTML, o CSS e o JavaScript. Essas três tecnologias são os pilares da web. E por que utilizá-las? Vamos entender alguns princípios e por que esse tripé é a melhor forma de iniciar na programação.

Desde que a internet se popularizou — por volta dos anos 1990 — entra e sai tecnologia. No entanto, três delas permaneceram ativas e fundamentais até hoje: HTML, CSS e JavaScript. Sabe aquelas tecnologias modernas que estão tomando conta do mundo hoje em dia, como React, Vue.js ou Angular? São famosas e populares, mas têm como base o JavaScript. Por isso chamamos de pilares da web. São fundamentais para o funcionamento do mundo digital e não há indícios de que isso vá mudar tão cedo.

Talvez você já tenha ouvido falar que Python é uma linguagem de programação fácil de aprender, ou que o "C++ dá mais dinheiro" ou que linguagem "X" é melhor. No final das contas, isso só deixa o iniciante perdido na hora de escolher o que estudar.

A primeira coisa que todo iniciante precisa entender é que, independente da linguagem de programação escolhida, a **lógica é a mesma**. O que é importante aprender é a **lógica de programação**. Isso significa que, a partir do momento em que você aprender bem a lógica, você vai ter menos dificuldades para aprender outras linguagens.

Resumindo: se você aprender bem o C++, você consegue em pouco tempo de estudo programar em Python ou PHP (ou qualquer outra), porque você tem o conhecimento estrutural de como funciona a lógica de qualquer programa.

Partindo do princípio de que a lógica é o mais importante na programação, por que começar com HTML, CSS e JavaScript?

Nos últimos anos, cerca de 99% das empresas deixaram de iniciar projetos em formato desktop — aqueles programas que precisam ser instalados no computador. Em vez disso, passaram a desenvolver aplicações que rodam diretamente em navegadores. E onde estão os navegadores hoje? Em praticamente todos os dispositivos com acesso à internet: computadores, notebooks, tablets, smartphones, televisores e até relógios inteligentes. Não importa se você usa Android, Windows, Linux ou macOS — todos esses sistemas oferecem acesso a navegadores modernos. **Todos**.

Por isso, aprender a programar para a web é uma escolha inteligente, atual e acessível. Isso significa que um único software web que você desenvolver vai rodar em qualquer dispositivo do seu cliente. Por que correr o risco de criar um programa que talvez não funcione em determinados aparelhos, simplesmente por não serem compatíveis com o Windows, por exemplo? A estratégia mais inteligente é investir no desenvolvimento de softwares para a web. E, nesse cenário, uma coisa é certa: seu projeto inevitavelmente contará com HTML, CSS e JavaScript.

Podemos concluir que, embora não seja obrigatório começar por HTML, CSS e JavaScript, essa é, sem dúvida, a escolha mais recomendada. Você vai aprender lógica de programação e vai construir aplicativos web que funcionarão em qualquer dispositivo com internet. Se um dia for necessário migrar para outra linguagem, já terá o conhecimento lógico estrutural para aprender rapidamente. Afinal, qual a melhor linguagem de programação? Essa é uma discussão recorrente, mas a resposta mais sensata é: a melhor

linguagem é aquela que abre as melhores oportunidades de crescimento profissional para você.

1.4 Frameworks

Se você está começando agora no mundo da programação, talvez ainda não conheça o termo "framework". Em resumo, um framework é um conjunto de ferramentas desenvolvidas por programadores e disponibilizadas gratuitamente para que outros desenvolvedores possam usá-las em seus próprios projetos.

Essas ferramentas geralmente resolvem tarefas comuns, como a criação de botões com cores, tamanhos e bordas já definidos. Com isso, você não precisa programar tudo do zero, o que poupa bastante tempo e acelera o desenvolvimento — e, convenhamos, todo mundo quer ganhar tempo, não é?

No nosso caso, vamos usar dois tipos de frameworks. Um se chama **Bootstrap**, que auxilia na criação de páginas web e assim ganharemos tempo com criação de botões, fontes, caixas, títulos etc. O outro que vamos utilizar será o **Chart.js** — que nos auxiliará a criar gráficos.

Precisamos necessariamente utilizar esses dois? Não. Existem vários outros frameworks similares a esses, como Tailwind, Buma, D3 etc. Até sugiro que você pesquise essas outras tecnologias, mas todas terão os mesmos princípios de uso e uma excelente documentação. Vamos utilizar essas pelo fato de terem grandes comunidades e, portanto, muita documentação disponível.

Fique tranquilo(a), nossa jornada está apenas começando! No próximo capítulo, vamos entender tudo isso de forma prática. Então, chega de teoria.

Capítulo 2
Estrutura e os primeiros códigos

Já temos nosso conjunto de ferramentas pronto para uso. Agora é hora de abrir nosso editor de texto e começar a codificar.

Desenvolver um software ou aplicativo é como planejar uma longa viagem. Você até sabe dirigir, mas é essencial definir a melhor rota, as paradas para refeições e descanso, o abastecimento e as estradas mais seguras. Ou seja, o planejamento é fundamental. A primeira etapa do projeto é estruturar nosso código.

Vamos conhecer as formas de organizar e, dependendo do seu projeto, você vai definir de um jeito ou outro. Vamos entender os princípios organizacionais para projetos de qualquer tamanho e, ao final deste capítulo, você vai perceber que "o céu é o limite".

2.1 Seja organizado(a)

É muito comum que projetos web sejam compostos por vários arquivos. Geralmente, eles incluem arquivos de código, imagens, talvez arquivos de vídeos ou música, PDFs e assim por diante. Dependendo do que será desenvolvido, o projeto pode exigir muitos arquivos diferentes.

Por isso, o ideal é sempre começar o projeto dentro de uma pasta. Ela pode estar em qualquer lugar do seu computador, mas é importante manter todos os arquivos organizados dentro dessa pasta principal. Ela é o que chamamos de raiz do projeto — ou seja, o ponto de partida, o "nível zero" da estrutura de arquivos.

Esse é apenas um termo técnico para designar o diretório onde o projeto começa, e não significa que o nome da pasta deva ser a raiz

do projeto. Na prática, você pode dar o nome que quiser (como `meu_site`, `dashboard_dados` etc.), desde que a estrutura esteja organizada.

Veja na imagem a seguir um exemplo visual de como essa organização pode ser feita:

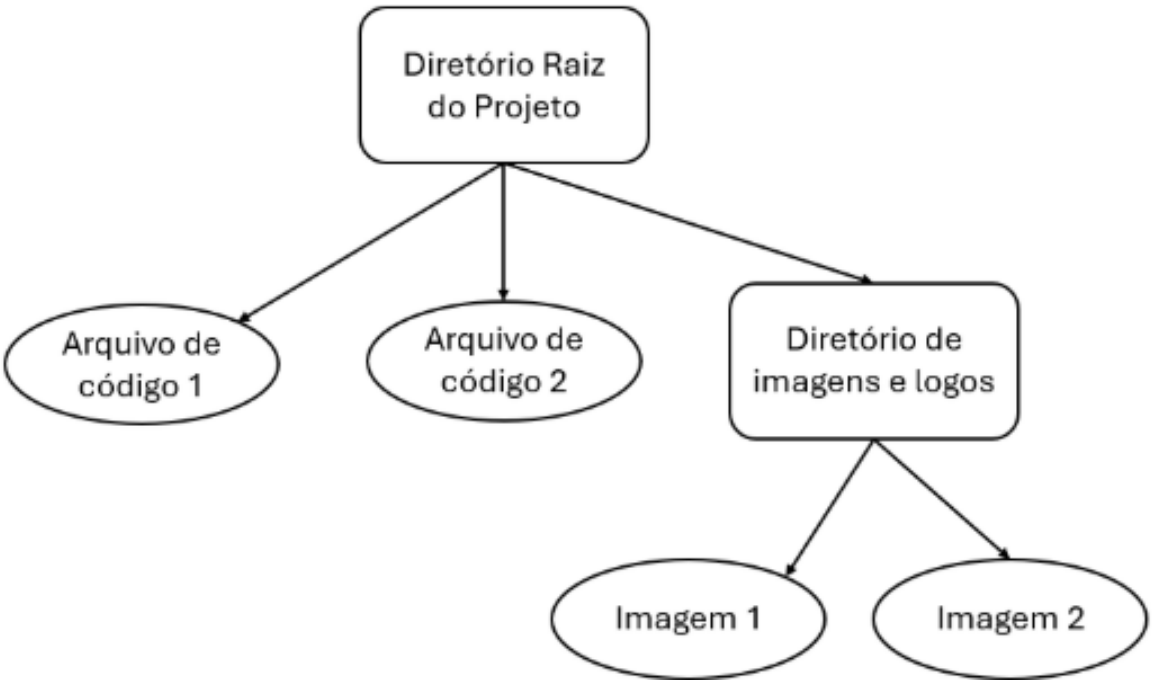


Figura 2.1: Exemplo de estrutura comum de um site

Essa hierarquia de arquivos ficaria algo similar a isso:

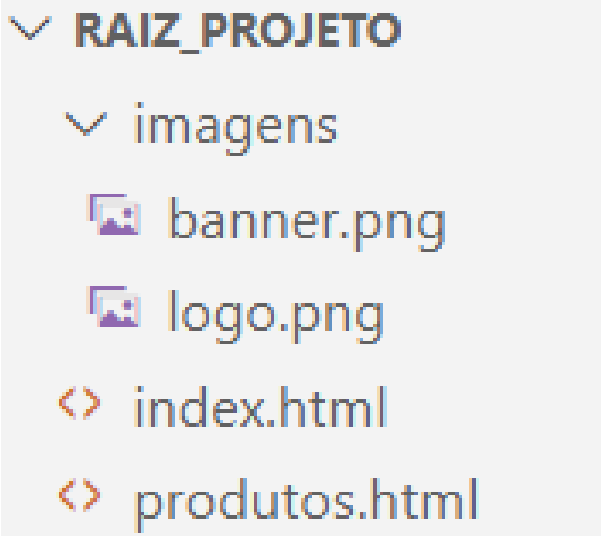


Figura 2.2: Exemplo de arquivos comuns em um site

Esse projeto web segue uma estrutura clara e organizada, similar à de qualquer sistema de pastas no computador. Temos um diretório principal chamado `raiz_projeto`, que contém:

- dois arquivos de código (`index.html` e `produtos.html`);
- uma pasta chamada `imagens`, onde estão armazenados os arquivos `banner.png` e `logo.png`.

Essa separação por tipo de conteúdo é uma ótima prática. Em vez de deixar todas as imagens espalhadas na pasta principal, agrupá-las em uma subpasta deixa o projeto mais limpo e fácil de entender. Caso você venha a trabalhar com vídeos, sons ou PDFs, o ideal é seguir o mesmo raciocínio e criar pastas como `videos`, `audios`, `documentos` etc.

Precisamos manter esse pensamento organizacional em todos os projetos, independentemente da linguagem utilizada — seja nos projetos deste livro ou em qualquer outro. Vamos aplicar essa organização também ao código.

2.2 A base da web

Começamos a transição entre organização de arquivos e organização de código. Toda página web precisa de três elementos fundamentais: HTML, CSS e JavaScript. Eles formam o que chamamos de os três pilares do desenvolvimento web:

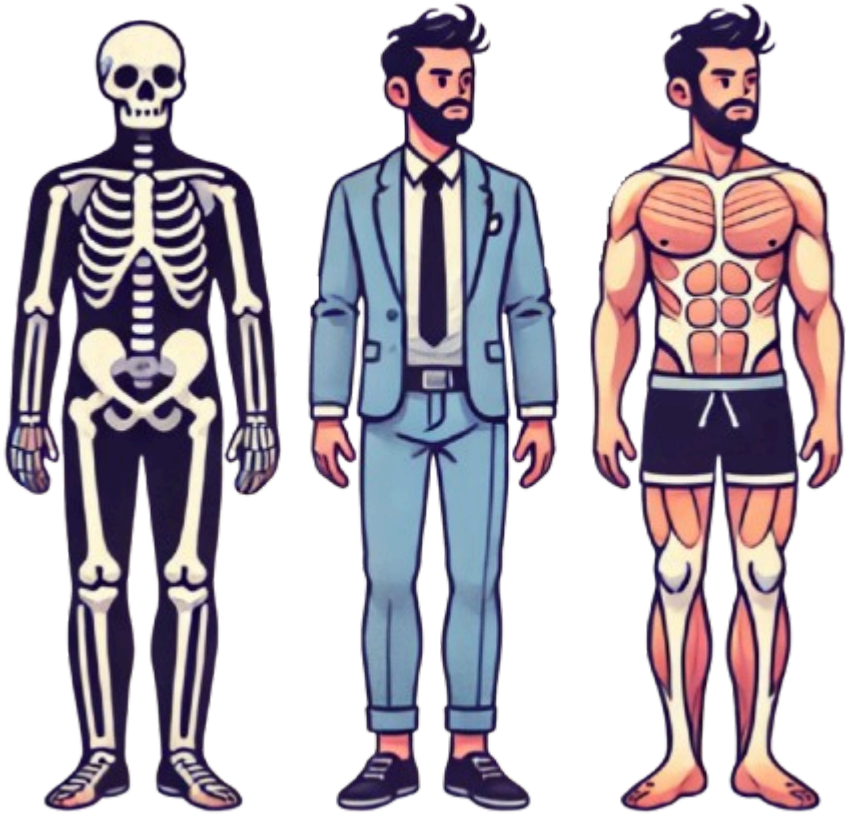


Figura 2.3: HTML, CSS e JavaScript

1. O HTML é o esqueleto. Formado por ossos que sustentam o corpo humano;
2. O CSS é a pele, cabelo, roupas, barba, relógio, anéis e qualquer outro acessório. Cria o visual da pessoa;

3. O JavaScript é a musculatura, que dá movimento e dinamismo ao corpo.

2.3 HTML

Com a analogia anterior em mente, vamos focar agora no esqueleto deste corpo humano. O **HTML é uma linguagem de marcação de etiquetas**, que permite criar documentos para a web. Usaremos com frequência três termos importantes: **tags**, **atributos** e **estrutura**.

Esses arquivos HTML são a base do nosso site e serão os primeiros a serem criados. Mas temos algumas boas práticas e convenções na hora de nomear esses arquivos, que ajudam tanto no funcionamento quanto na organização do projeto.

Convenção de nomenclatura dos arquivos

As recomendações a seguir valem principalmente para arquivos HTML, mas são boas práticas que também se aplicam a arquivos CSS, JS e outros recursos da web.

- **Evite espaços, acentos ou símbolos no nome do arquivo:** Por exemplo, se você tiver uma página que mostrará os serviços rápidos de uma empresa, é comum usar um nome que indique sua função, como `servicos-rapidos.html`. Essa escolha melhora a legibilidade do nome e evita problemas de leitura nos navegadores.
- **Use letras minúsculas:** Essa prática padroniza os arquivos e evita que links pareçam "gritar" com o usuário, como em `www.espaco.dev/PRODUTOS.html`.
- **Utilize a extensão correta:** Para arquivos HTML, utilize a extensão `.html`. No Windows, atenção para não salvar o

arquivo como `index.html.txt`, o que pode ocorrer dependendo do editor de texto.

Tags HTML

Agora que sabemos como nomear e organizar os arquivos, vamos entender o que são as **tags HTML**. As tags são os elementos que estruturam uma página web. São chamadas de "etiquetas" porque marcam trechos do conteúdo indicando o que eles representam (um título, um parágrafo, um link etc.).

As tags HTML são sempre escritas em pares (um inicial antes do conteúdo e um final depois do conteúdo):

```
<tag>
    Conteúdo aqui
</tag>
```

A tag de abertura contém apenas o nome do elemento (como `<p>`), e a de fechamento contém uma barra antes do nome (como `</p>`).

Além disso, as tags podem conter atributos — características adicionais que definem comportamentos ou propriedades. Os atributos são sempre escritos na tag de abertura, com nome e valor entre aspas, como neste exemplo:

```
<tag atributo1="valor1" atributo2="valor2">
    Conteúdo aqui
</tag>
```

Veja uma demonstração de código HTML com uma tag de título e um link:

```
<html>
    <body>
        <h1>Título do meu site</h1>
```

```
        <a href="http://espaco.dev" target="_blank">
            Clique aqui para ir para o site
        </a>
    </body>
</html>
```

Esse código gera o seguinte site:

Figura 2.4: Resultado de um HTML básico

Nessa imagem temos algumas tags. Vamos analisar o que elas fazem:

- `html` – Tag que informa ao navegador que esse documento se trata de um documento de site. Observe que essa tag se abre antes de tudo e fecha depois de tudo. Essa tag sempre se comportará assim. Afinal, tudo dentro deste arquivo fará parte do que será nosso documento web.
- `body` – Tag que abrange todo o conteúdo visível da página no navegador.
- `h1` – Tag que cria o título. O título está sem atributos também.
- `a` – Tag que cria o link e possui dois atributos. O atributo `href` informa para qual endereço o usuário será destinado quando clicar e o atributo `target` define a forma, no caso do valor `_blank` a página é aberta em uma nova aba no navegador.

Com esse primeiro exemplo, já podemos compreender mais alguns aspectos do HTML. É possível agrupar tags? Colocar uma dentro da outra? Sim — e isso é algo bastante comum. O mais importante é lembrar que elas sempre devem ser fechadas na ordem inversa à de abertura:

```
<elemento>
    <subelemento>
        Isso está errado
    </elemento>
</subelemento>

<elemento>
    <subelemento>
        Isso está correto!
    </subelemento>
</elemento>
```

Indentação do código

Talvez você não tenha notado, ou até percebeu, mas pode ser que achou que isso era parte do código. Então observe:

```
1   <!DOCTYPE html>
2   <html lang="en">
3   <head>
4       <meta charset="UTF-8">
5       <meta name="viewport" content="width=device-width, initial-scale=1.0">
6       <title>Meu site</title>
7   </head>
8   <body>
9       <div class="container">
10          <h1>Bem-vindo ao Meu Site</h1>
11          <p>Acesse o conteúdo completo clicando no link a seguir:</p>
12          <a href="https://espaco.dev" target="_blank" class="botao">
13              Visite o Espaço Dev
14          </a>
15      </div>
16  </body>
17  </html>
```

Figura 2.5: Código indentado

Quero que você perceba que os códigos têm uma estrutura hierárquica e que, a cada tag aberta, adiciona-se um espaço adicional na linha seguinte, até que a tag de fechamento apareça

No exemplo da imagem acima, observe a tag `<div>`. Ela é aberta e depois existem algumas outras tags dentro dela, como `<h1>`, `<p>`, `<a>`. Somente depois que todas essas tags se fecham é que a tag `<div>` também se fecha.

Isso nos mostra que, além de uma estrutura hierárquica de tags, há uma organização visual no código graças a esses espaçamentos. Embora não sejam obrigatórios, é importante notar que esses espaços criam uma **organização visual incrível** e por isso o uso dessa técnica se torna mais que recomendado.

A indentação pode ser feita de duas formas:

- Usando a tecla TAB;
- Ou usando espaços (geralmente 2 ou 4 espaços por nível).

O mais importante é manter um padrão. Você pode escolher o que preferir, mas a recomendação geral é **usar uma tecla TAB**, pois facilita a leitura e evita quebras de formatação em diferentes editores ou sistemas.

Se você não adicionar esses espaçamentos a cada nova linha do seu código, ele ainda funcionará — pois, como mencionei, o mais importante é que o código esteja digitado corretamente. No entanto, a organização proporcionada por esses espaços é fundamental para facilitar o seu entendimento, e também para qualquer outra pessoa que, futuramente, precise acessar ou modificar o seu código.

2.4 CSS

Também precisamos explorar este pilar da web, que, como vimos anteriormente, é responsável por tornar as páginas web mais bonitas e atraentes para quem as visualiza.

Assim, vamos analisar o mesmo código HTML, porém com a adição de um código CSS e observar a mágica:

HTML:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Meu site</title>
    <link rel="stylesheet" href="estilos.css">
</head>
<body>
    <div class="container">
        <h1>Bem-vindo ao Meu Site</h1>
        <p>Acesse o conteúdo completo clicando no link a seguir:</p>
        <a href="https://espaco.dev" target="_blank" class="botao">
            Visite o Espaço Dev
        </a>
    </div>
</body>
</html>
```

CSS (em outro arquivo chamado `estilos.css`):

```
body {
    font-family: "Arial, sans-serif";
    background-color: #f4f4f9;
    margin: 0;
    padding: 20px;
}
```

```css
.container {
    text-align: center;
    background-color: #fff;
    padding: 20px;
}

h1 {
    color: #333;
}

.botao {
    margin: 10px;
    text-decoration: none;
    color: #fff;
    padding: 10px 20px;
    border-radius: 10px;
}

.botao:hover {
    background-color: #0056b3;
}
```

Nesse código, o HTML tem uma linha a mais:

```html
<link rel="stylesheet" href="estilos.css">
```

Essa linha é responsável por adicionar ao projeto um outro arquivo — neste caso, o `estilos.css`. Esse arquivo contém o código CSS que será aplicado ao HTML. O CSS possui uma estrutura diferente do HTML, pois não é composto por tags.

O formato básico de um bloco CSS consiste em um seletor, seguido por um par de chaves `{ }`. Dentro dessas chaves são definidas propriedades e seus respectivos valores, separados por dois-pontos `:` e finalizados com ponto e vírgula `;`.

```css
seletor {
    propriedade: valor;
}
```

Por exemplo, no arquivo `estilos.css` temos alguns blocos de conteúdo:

`body{ }` – aplica estilos diretamente ao elemento `<body>` do HTML:

- `font-family: Arial, sans-serif;` – Define a fonte de todo o conteúdo da página com Arial, sem serifa;
- `background-color: #f4f4f9;` – Define a cor do fundo da página utilizando um código hexadecimal;
- `margin: 0px;` – Remove a margem padrão entre o navegador e o conteúdo da página;
- `padding: 20px;` – Define um espaçamento interno de 20 pixels para todo o conteúdo da página;
- `.container { }` – Aplica estilos a qualquer elemento que possua a **classe** `container`.

Uma "classe" no HTML é um atributo usado para agrupar elementos com o mesmo estilo. No exemplo `<div class="container">`, estamos dizendo que essa `<div>` pertence ao grupo chamado "container", e isso permite aplicar estilos CSS a todos os elementos com essa mesma classe.

Esse seletor é usado, por exemplo, com uma tag como `<div class="container">`.

- `text-align: center;` – Centraliza todo o texto contido dentro da `<div>`;
- `background-color: #fff;` – Define o fundo branco da `<div>`;
- `padding: 20px;` – Adiciona espaçamento interno de 20 pixels ao conteúdo da `<div>`;
- `h1{ }` – Aplica estilos a todas as tags `<h1>`;
- `color: #333;` – Define a cor do texto do título utilizando o código hexadecimal `#333`;
- `.botao{ }` – Aplica estilos a qualquer elemento que possua a **classe** `botao`;

- `margin: 10px;` – Cria um espaçamento externo de 10 pixels em relação aos elementos ao redor;
- `text-decoration: none;` – Remove qualquer decoração de texto (como sublinhados em links);
- `background-color: #007BFF;` – Aplica um tom de azul de fundo ao botão;
- `padding: 10px 20px;` – Adiciona 10 pixels de espaçamento vertical e 20 pixels horizontal dentro do botão;
- `border-radius: 10px;` – Arredonda os cantos do botão;
- `.botao:hover{ }` – Aplica estilos **somente quando o cursor estiver sobre o botão** (efeito hover);
- `background-color: #0056b3;` – Define uma tonalidade de azul mais escura para destacar visualmente o botão ao passar o mouse.

Regras e padrões no CSS

Assim como vimos no HTML, o CSS também segue algumas boas práticas de escrita:

- **Indentação:** Não é obrigatória, mas facilita muito a leitura. Assim como no HTML, recomenda-se o uso de 2 ou 4 espaços por nível.
- **Propriedades terminam com ponto e vírgula** `;` .
- **Cada propriedade usa dois-pontos** `:` entre nome e valor, diferente do HTML, onde usamos `=` nos atributos.
- **Seletor de classe usa ponto** (.) antes do nome, como `.botao` , e é aplicado via `class="botao"` no HTML.

Agora observe o resultado após todo esse código CSS adicionado.

Figura 2.6: À esquerda, HTML puro. À direita, HTML com CSS

A diferença é muito grande, concorda? O HTML puro fica estrutural, sem estilo visual. Com o CSS, o site ganha **vida e coloração** e fica mais agradável, funcional. Como vimos, todas as propriedades explicadas anteriormente foram aplicadas ao exemplo da imagem à direita.

2.5 Bootstrap — Framework de CSS

Agora que você tem uma grande noção do poder do CSS, é hora de conhecer o framework que vai nos auxiliar nesta jornada. Como vimos, framework é algo que é desenvolvido e disponibilizado na internet com o intuito de auxiliar outros programadores.

No tópico anterior, talvez você tenha pensado que o CSS é bacana, mas é muito código para poucos resultados. Ou que teria que decorar muitas propriedades para conseguir aplicar no seu HTML. O intuito desta seção é justamente analisar quais ferramentas nos trazem um compilado pronto que ajuda e acelera demais esse processo.

Para uma nova comparação, observe o código e a imagem a seguir, compostos apenas em HTML, aos quais foi adicionado o framework de CSS Bootstrap.

```
<html lang="pt-br">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Meu site</title>
    <!-- Linha a seguir para instalar Bootstrap CSS -->
    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.
css" rel="stylesheet" />
</head>
<body class="bg-light">
    <div class="container mt-5">
        <div class="card shadow-lg text-center p-4">
            <h1 class="mb-3">Bem-vindo ao Meu Site</h1>
            <p class="mb-4">Acesse o conteúdo completo clicando no link a
seguir:</p>
            <a href="https://espaco.dev" target="_blank" class="btn btn-
primary">
                Visite o Espaço Dev
            </a>
        </div>
    </div>
</body>
</html>
```

Figura 2.7: Bootstrap - Framework de CSS

Olha que coisa incrível. Sem adicionar nenhum código CSS, o nosso site foi estilizado como se tivesse muito código CSS nele (e realmente tem). Nós só reaproveitamos o CSS que já é criado do framework e as propriedades são automaticamente aplicadas no nosso site. Na verdade, esse exemplo ficou até mais bonito do que o código CSS feito manualmente.

O que temos que destacar no código que gerou esse site:

- Código da linha `<link href=...` : Essa linha é responsável por instalar o Bootstrap no nosso código HTML. Veja que ela é bem parecida com a forma anterior de chamar um outro código CSS. A diferença é que o link é bem maior, pois estamos chamando um código pronto existente na internet.
- Código das linhas dentro da primeira `<div>` : Essas linhas têm várias classes inseridas junto com as tags e são essas classes que já estão prontas para usar. Por exemplo, o `class="container mt-5"` já cria um distanciamento das bordas e um distanciamento do topo de 5. A `class="bg-light"` já cria automaticamente um plano de fundo claro para o `body` do seu site. A `class="card shadow-lg text-center p-4"` cria esse estilo cartão (por isso, `card`) para o conteúdo com um sombreamento e com um espaçamento interno tamanho 4.

Talvez você esteja se perguntando como aprender esses códigos, e aqui vai outra vantagem, em comparação ao CSS "cru" e escrito manualmente: o framework Bootstrap possui uma documentação gratuita, muito fácil de ler e aplicar. Conforme você o utiliza mais, percebe que as nomenclaturas são intuitivas e a facilidade que isso proporciona. Por exemplo, para criar uma tabela, a classe é “table”; para criar um botão, a classe é “btn”, e assim por diante.

O manual está disponível em https://getbootstrap.com/.

Ah! Eu ia deixar para citar a melhor vantagem do framework por último: todos os seus componentes são **responsivos**. Isso quer dizer que são automaticamente adaptáveis a qualquer tipo de tela.

Observe o mesmo código em uma tela de 360 x 740 (360 de largura por 740 de altura), similar à tela de um celular Samsung Galaxy S8+:

Figura 2.8: Site adaptado para tela pequena

E agora veja o código em um monitor de notebook de resolução de 1426 x 760:

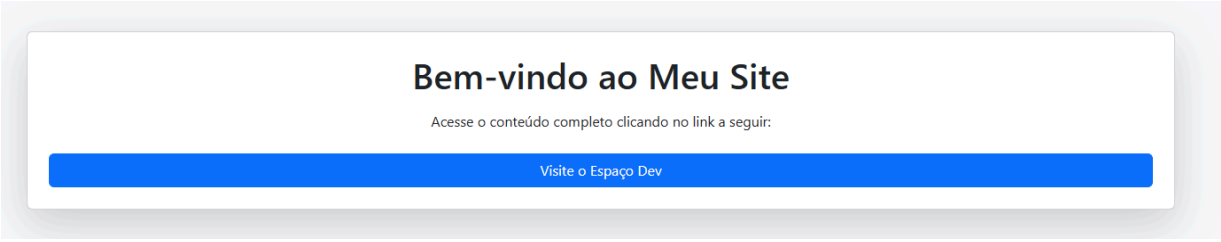


Figura 2.9: Site adaptado para tela grande

O código se adaptou automaticamente sem ser necessário alterar nenhuma linha. O framework Bootstrap tem o código pré-

configurado para que seus componentes sejam dimensionados conforme o tamanho da tela do dispositivo do seu usuário.

Encerrando esta etapa...

Neste capítulo, organizamos os arquivos, entendemos a estrutura de um projeto web, mergulhamos no HTML e conhecemos o CSS como ferramenta para dar vida ao nosso layout. Percebemos como uma página web se forma a partir da união dessas linguagens — com o HTML como esqueleto e o CSS como aparência. Mas ainda falta algo: **o movimento**.

No próximo capítulo, vamos conhecer o JavaScript, a linguagem de programação que dá vida aos sites, permite interação com os usuários e é o que realmente transforma um layout em uma aplicação.

Parte II — Explorando o poder do JavaScript

Agora que você entende o processo e as ferramentas essenciais para criar Dashboards, é hora de mergulhar no coração da web interativa: o JavaScript.

Nesta parte, vamos percorrer uma jornada que vai desde os fundamentos da linguagem até conceitos mais avançados. O objetivo é que você se sinta confortável para interagir com elementos da página e preparar os dados para visualizações impactantes.

Você aprenderá:

- A sintaxe básica da linguagem e como ela funciona no navegador.
- Funções, estruturas de repetição e condicionais.
- Como organizar melhor seu código com funções modernas e boas práticas.
- O papel do JavaScript na comunicação com servidores e APIs.
- Como integrar seus scripts com o HTML e tornar suas páginas dinâmicas.

Mesmo que você nunca tenha programado com JavaScript, esta parte foi feita para você. E se já tem alguma familiaridade, verá exemplos práticos que irão ampliar sua visão e seu domínio da linguagem.

Vamos explorar o que torna o JavaScript uma ferramenta tão poderosa para a criação de Dashboards inteligentes e responsivos!

Capítulo 3
Bem-vindo ao JavaScript

Aprendemos a estruturar e estilizar nossas páginas. Chegou a hora de dar **vida e inteligência** ao nosso projeto. Neste capítulo, vamos conhecer o **JavaScript**, a linguagem que movimenta a web. Ele será a base de todas as funcionalidades e interações que faremos daqui em diante — inclusive os nossos Dashboards dinâmicos que serão criados no navegador.

Além disso, como este livro também tem o objetivo de ensinar a programar, vamos usar o JavaScript para aprender os principais conceitos de **lógica de programação**. Você não precisa conhecer outras linguagens para acompanhar este conteúdo. O JavaScript é uma excelente primeira linguagem — moderna, popular e aplicável em diversas áreas da tecnologia.

Veja os gráficos a seguir divulgados pelo GitHub:

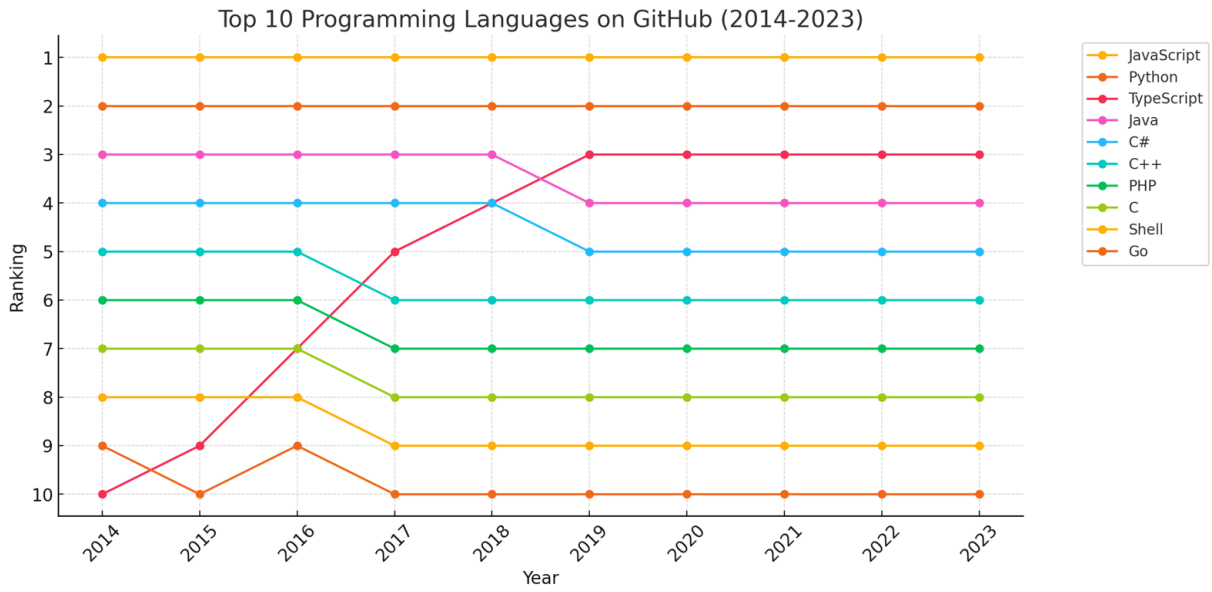


Figura 3.1: As 10 linguagens de programação mais usadas no GitHub de 2014 a 2023

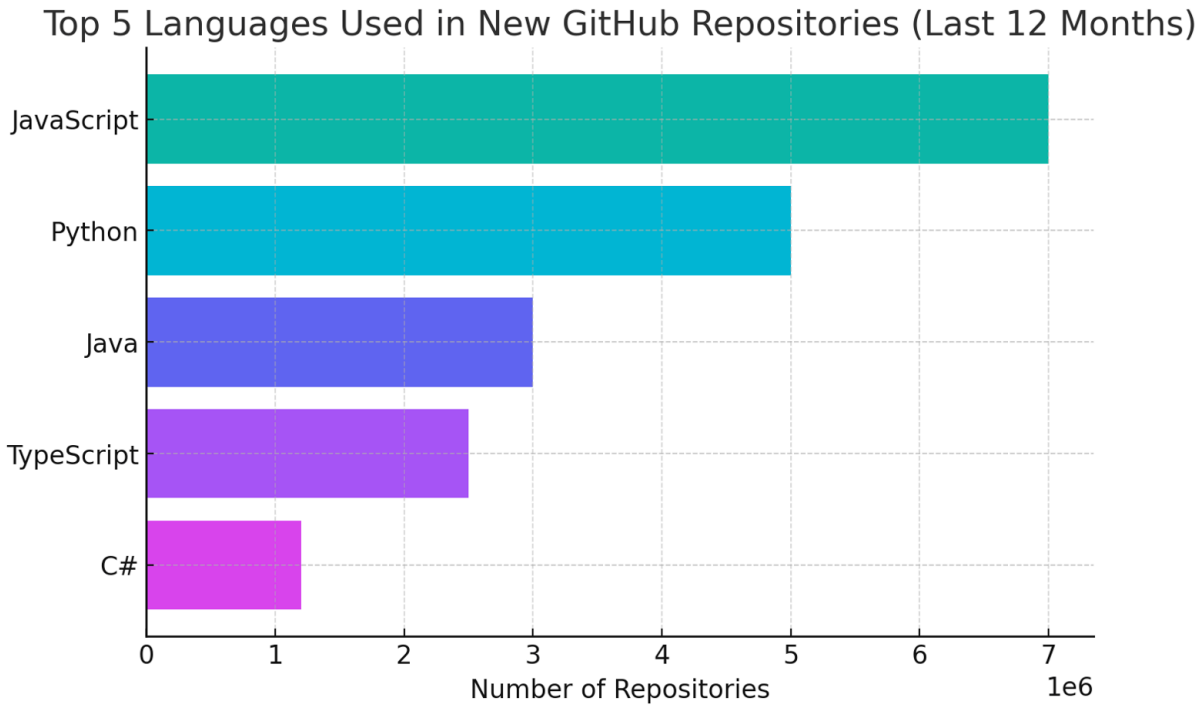


Figura 3.2: As 5 linguagens mais comuns encontradas nos repositórios do GitHub em 2023 - Nºs em milhões

Como vimos, o JavaScript se mantém, há anos, como uma das linguagens mais populares e aplicadas no mundo. Ele é a base de frameworks como React.js, Vue.js e Angular.js, além de ser amplamente utilizado no back-end com Node.js e até mesmo em bancos de dados como o MongoDB.

> Aprender JavaScript é abrir portas para diversas possibilidades profissionais. Mesmo que você venha a trabalhar com outras tecnologias no futuro, os fundamentos que aprenderá aqui servirão para qualquer caminho.

3.1 O que é JavaScript?

Se você chegou até aqui, talvez já tenha uma boa noção do que é essa linguagem. Mas vamos deixar bem claro, especialmente para quem está começando do zero:

JavaScript é uma linguagem de programação usada principalmente para dar **dinamismo** aos sites e aplicativos web. Ela interage com o HTML e o CSS, adicionando comportamentos, reações a cliques, validações de formulários e muito mais.

Ela é uma **linguagem interpretada**, ou seja, seu código é lido e executado diretamente pelo navegador, **linha por linha**, sem a necessidade de uma etapa de compilação como acontece em linguagens como C ou Java.

> **Linguagens interpretadas** (como JavaScript): executam o código em tempo real, linha por linha.
>
> **Linguagens compiladas** (como C, Java): o código inteiro precisa estar pronto e correto para ser transformado em um arquivo executável.

Não há uma que seja mais adequada que a outra — são apenas **formas diferentes de funcionamento**.

JavaScript no front-end e back-end

O JavaScript é tão versátil que pode ser usado nos dois lados de um aplicativo:

- **Front-end:** é a parte da aplicação com a qual o usuário interage — as telas, botões, formulários, menus, gráficos.
- **Back-end:** é onde ficam as regras de negócio, os bancos de dados e onde os dados do usuário são processados e armazenados.

Por exemplo, ao criar uma conta em uma rede social, você digita seus dados em um formulário (front-end) e, ao clicar em

"Cadastrar", essas informações são enviadas para o servidor (back-end).

> Neste livro, focaremos no **JavaScript no front-end**, ou seja, **tudo será executado diretamente no navegador** do usuário. Essa abordagem é leve, rápida e perfeita para criação de Dashboards interativos sem a necessidade de servidores complexos.

3.2 Como escrever o código em JS

Existem duas formas de adicionar JavaScript em qualquer projeto web. A primeira é colocando uma tag `<script>` dentro do próprio arquivo HTML e escrever seus códigos JS dentro deste mesmo arquivo:

```
    <!-- Seu HTML vem acontecendo aqui -->
    </div>
        <script>
            //Aí entra uma tag de script e um código JS dentro dessa tag assim:
            alert("Olá mundo!");
        </script>
    </body>
</html>
```

E a segunda maneira é fazer referência a outro arquivo com outra extensão, assim:

```
        <!-- Seu HTML acontecendo aqui -->
        </div>
        <!-- Uma tag de script que faz referência a outro arquivo -->
        <script type="text/javascript" src="scripts.js"></script>
    </body>
</html>
```

Observe que o arquivo `index.html` contém apenas uma tag `<script>` com o atributo `src`, que faz referência a outro arquivo. Ou seja, o código JavaScript não está no próprio HTML, mas é carregado externamente. Para isso, é necessário indicar corretamente o nome do arquivo no atributo src da tag `<script>`.

Mas qual dos dois modos é mais adequado para utilizarmos o JavaScript? Vamos analisar, pois essa compreensão faz muita diferença na formação de programadores e programadoras com boas práticas de organização e estrutura de código.

No primeiro formato, dentro de um arquivo `.html` você terá códigos HTML e JavaScript. Não há nenhum problema nisso, mas, do ponto de vista de organização, note que em um mesmo arquivo você terá duas linguagens distintas. Podemos fazer uma analogia com um texto qualquer: imagine que você está lendo um texto em português e, no meio da história, o texto muda para inglês e, depois de alguns parágrafos, volta para português. Mesmo que você saiba as duas línguas, é estranho.

Já na segunda forma, o projeto fica mais organizado. Fica claro que há arquivos separados para diferentes linguagens. Outro ponto importante é que o JS é a linguagem responsável por trazer dinamismo às páginas; por isso, ao visualizar a estrutura do projeto, é fácil identificar onde estão as funcionalidades e onde está o layout. Portanto, em projetos mais robustos — com muitas páginas HTML e diversos arquivos JS —, o resultado será um sistema bem estruturado, muito mais organizado e profissional.

Vamos usar sempre essa segunda abordagem. Utilizaremos arquivos `.html` para colocar as informações na página e os arquivos `.js` para criar o dinamismo.

Vamos aproveitar esse momento organizacional para algumas dicas adicionais:

- Seu projeto pode ter mais de um arquivo `.js`, fazendo suborganizações. Vamos supor que seu aplicativo web, em

momentos distintos, guarde a localização do usuário e também use a câmera do celular ou notebook. Você pode colocar as funcionalidades da geolocalização em um arquivo `.js` e as da câmera em outro arquivo `.js`.
- Onde é melhor colocar a tag `<script>`? Há diversas recomendações na internet — dentro da tag `<head>`, no início do `<body>`... mas aqui vai minha dica, baseada em prática: sempre coloque a tag `<script>` no final do arquivo `.html`, logo antes de fechar a tag `</body>`. Isso garante que o conteúdo da página (HTML) seja carregado primeiro, antes das funcionalidades em JS. Assim, o que é visível aparece mais rápido para o usuário, enquanto os scripts são carregados em seguida. Essa prática melhora o tempo de carregamento da página — mesmo que em milissegundos — e ajuda a manter seu projeto mais eficiente. Adotar esse hábito é uma forma simples e eficaz de otimizar seus projetos.

3.3 Regras básicas do JavaScript

Em meio aos nossos códigos, precisamos entender a liberdade que temos dentro dos arquivos. Então temos algumas regras iniciais:

- Os espaços em branco e as novas linhas não são considerados. Se você pular duas linhas de um código para o outro não fará nenhuma diferença na execução do seu programa. Isso ocorre tanto no HTML quanto no JS.

- *Case sensitive*: Esse termo quer dizer que o JavaScript faz diferença entre letras maiúsculas e minúsculas. Vamos supor exista um código que guarda a somatória de alguns valores. Esse código se chama "soma". Se em algum momento do nosso programa usarmos o nome "Soma" não estamos falando da mesma coisa. Para o JS "soma" é diferente de "Soma", que é diferente de "sOma" que é diferente de "SOMA".

- Comentários: São blocos de informações que podemos adicionar em qualquer trecho no meio do nosso código. Essas informações geralmente servem para explicar em linguagem natural alguma funcionalidade complexa que foi feita naquele trecho do código. Para adicionar um comentário no seu código basta adicionar duas barras (`//`) para que tudo depois delas seja ignorado naquela linha. Ou então, se seu comentário for maior que uma linha, você pode colocar todo seu texto entre `/*` e `*/` . Tudo que estiver entre esses caracteres será ignorado pelo programa.

```
<script>
    alert("Oi!"); //Comentário simples de uma única linha
    /* Esse trecho é um exemplo de comentário que será ignorado pelo
navegador, mas pode ser informativo para o programador. */
</script>
```

> Essa prática de comentar o código é universal e existe em praticamente todas as linguagens de programação. Os símbolos podem variar — enquanto em JavaScript usamos `//` e `/* ... */` , em linguagens como Python e Shell Script o símbolo é o `#` . Já no HTML, os comentários são escritos assim: `<!-- comentário -->` . Eles também não aparecem na tela do usuário e servem para documentar o código.

- Palavras reservadas: Em vários casos, vamos criar nomes para as coisas que fazemos no código e algumas palavras não poderemos usar pois são próprias do JS, tais como: `break` , `case` , `catch` , `continue` , `default` , `let` , `delete` , `do` , `else` , `finally` , `for` , `function` , `in` , `new` , `return` , `if` , `switch` , `while` , `this` , `try` , `var` , `void` etc.

3.4 Tipo de dados

Qualquer site ou plataforma digital manipula diversos tipos de informações. Essas informações são divididas nestes 4 tipos:

- Texto: Representa uma sequência de um ou mais caracteres. Esse tipo de informação é colocado entre aspas. Ex.: nome.
- Inteiro: Representa valor numérico. Positivos ou negativos. Sem casas decimais. Ex.: idade.
- Real: Representa valor numérico. Positivos ou negativos. Com casas decimais. Ex.: altura ou valor monetário.
- Lógico: Representa 2 alternativas: Sim ou não. Verdadeiro ou Falso. 0 ou 1. Também são chamados de valores booleanos.

Sempre que você digita alguma informação, e também quando o programa está fazendo algum cálculo ou exibe alguma informação para você, esse valor ou informação foi gerado usando um desses tipos de dados que citamos e estava guardado em uma variável.

3.5 Variáveis

Uma variável é um espaço na memória que, como seu próprio nome indica, pode mudar de conteúdo ao longo da execução de uma página. Usamos essas variáveis para guardar os dados anteriormente citados. A seguir, vamos executar um programa bem simples que está apenas guardando algumas variáveis e exibindo:

- Arquivo `index.html` :

```
<html>
    <head>
        <title>Meu Site</title>
    </head>
    <body>
        <h1>Meu site</h1>
        <script src="scripts.js"></script>
```

```
    </body>
</html>
```

- Arquivo `scripts.js` :

```
var cliente = 'João';
var idade = 34;
var preco = 9.9;
var estaAtivo = true;

console.log(cliente);
console.log(idade);
console.log(preco);
console.log(estaAtivo);
```

Esse programa exibirá:

Figura 3.3: Programa simples com variáveis no console

Nesta imagem, vemos o navegador exibindo o resultado do nosso programa, que está sendo executado a partir dos arquivos `index.html` e `scripts.js` . Como já vimos, o arquivo HTML é responsável por estruturar a página e incluir o script, enquanto o arquivo JavaScript contém as instruções que realmente fazem algo acontecer.

A partir de agora, durante nosso aprendizado da linguagem JavaScript, o foco será sempre no conteúdo do arquivo `scripts.js` ,

pois é nele que escreveremos todo o código da linguagem. Para manter os exemplos mais limpos e objetivos, vamos sempre presumir que existe um arquivo `index.html` básico já configurado para carregar o script. Assim, nos próximos trechos, mostraremos o conteúdo do JavaScript diretamente (em blocos de código como mostrado), e as imagens trarão apenas a saída gerada no navegador, para facilitar sua leitura.

Vamos entender os códigos: criamos uma variável para cada tipo de dado através do var `nome_variavel = valor_da_variavel` . Cada dado foi armazenado em sua respectiva variável de acordo com o tipo. A variável do tipo texto é declarada entre aspas; a do tipo numérico, sem aspas; e, nos casos em que há separador decimal, utiliza-se o ponto (.) — nunca a vírgula, que possui outras funções na programação, como veremos adiante.

Por fim, temos o código `console.log()` , que exibe, na ferramenta do desenvolvedor, os valores passados entre seus parênteses. Ao inserirmos variáveis como argumento, podemos visualizar seus conteúdos diretamente no console do navegador. Utilizaremos bastante esse comando para testar o funcionamento do código e verificar quais dados estão armazenados nas variáveis em diferentes momentos da execução. É um recurso prático e essencial para quem está desenvolvendo aplicações web.

Declaração, atribuição e inicialização

Declarar uma variável significa criá-la. Para isso, usamos uma palavra reservada, como `var` . Escrevemos essa palavra-chave e, em seguida, o nome que queremos dar à nossa variável. Não devemos utilizar espaços nem caracteres especiais. O ponto e vírgula (;), por sua vez, indica que encerramos o código desta linha.

```
// Forma correta
var cliente;
var preco;
var idade;
```

```
// Forma incorreta
var o ano
var preço
var número
```

Podemos atribuir vários tipos de valores às variáveis em partes distintas do programa. Isso é feito por meio do operador de atribuição, representado pelo símbolo = (igual). A variável que receberá o valor deve estar à esquerda, e o valor atribuído deve estar à direita.

```
// Aqui declaramos a variável
var cliente;
var preco;
var idade;


// Aqui atribuímos valores às variáveis
cliente = "Lucas";
preco = 99.90;
idade = 34;
```

Também é possível declarar uma variável e inicializá-la com um valor desta forma:

```
// Declaração e inicialização
var cliente = "Lucas";
var preco = 99.90;
var idade = 34;
```

Vamos executar um exemplo um pouco maior para observar que em diferentes lugares do código é possível mudar o valor de uma variável. É importante também notar no código a seguir que cada variável só pode ser declarada apenas uma vez no mesmo programa:

```
//Declaração e inicialização
var cliente = "Lucas";
var preco = 99.90;
var idade = 34;
```

```
//Apresentação das variáveis no console
console.log(cliente);
console.log(preco);
console.log(idade);

//Alteração de valores das variáveis já existentes
cliente = "João";
preco = 45.40;
idade = 35;

//Apresentação das mesmas variáveis no console com novo valor
console.log(cliente);
console.log(preco);
console.log(idade);
```

Figura 3.4: Programa JS com variáveis

3.6 Operações

Agora que sabemos como atribuir valores às variáveis, podemos fazer atribuições a partir de contas matemáticas, comparações ou até condições que acontecem no decorrer da execução do programa. Faremos isso com exemplos para facilitar o entendimento. No decorrer desta seção do livro vamos conhecer diversos operadores. Começaremos pelos aritméticos.

Operadores aritméticos

São utilizados para obter resultados numéricos. Os símbolos para os operadores aritméticos são:

- Soma, representado por `+`;
- Subtração, representado por `-`;
- Multiplicação, representado por `*`;
- Divisão, representado por `/` (não confunda com a `\`, que haverá erro no seu código);
- Exponenciação, representada por `**`;
- Módulo (resto de uma divisão), representado por `%`.

Operações básicas

Com esses operadores já podemos codificar alguns cálculos matemáticos para treinar e ver seu resultado com o comando `console.log`:

```
var x = 5;
var y = 10;
var z = 20;

//Somar dois valores e colocar o resultado em nova variável
var resultado_soma = x + y;

//Subtrair dois valores e colocar o resultado em nova variável
var resultado_subtracao = x - y;

//Multiplicar dois valores e colocar o resultado em nova variável
var resultado_multi = x * y;
```

```
//Dividir dois valores e colocar o resultado em nova variável
var resultado_divisao = z / y;

//Exponencial de dois valores e colocar o resultado em nova variável
var resultado_exponencial = x ** 2;

//Resto da divisão de dois valores e colocar o resultado em nova variável
var resultado_resto = z % x;

//Exibir os resultados no console do navegador
console.log(resultado_soma);
console.log(resultado_subtracao);
console.log(resultado_multi);
console.log(resultado_divisao);
console.log(resultado_exponencial);
console.log(resultado_resto);
```

Esse código trará um resultado assim:

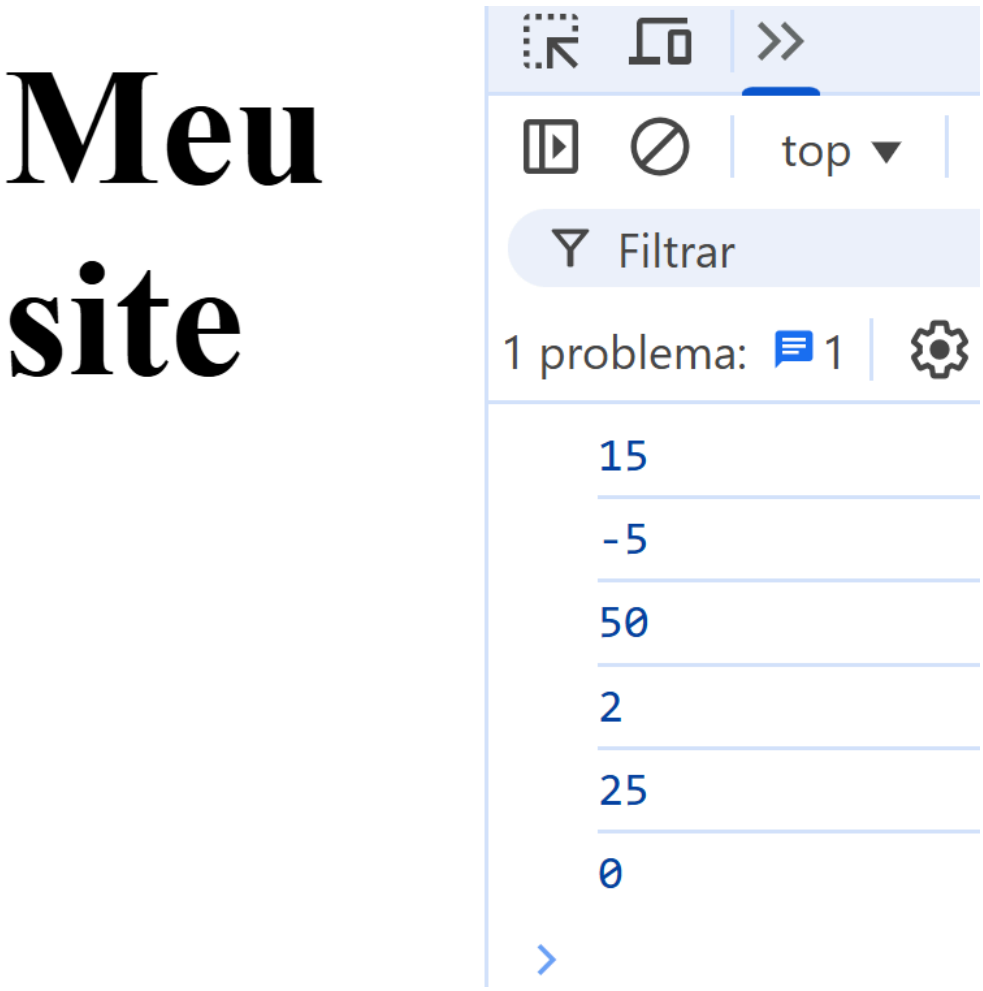


Figura 3.5: Operações matemáticas com JS

Cada variável chega nas linhas do comando `console.log` com um valor resultante de uma conta e são exibidos no console do navegador.

3.7 Prompt, console e alert

Como vimos até aqui o `console.log()` é um comando de interação com a pessoa que está desenvolvendo a página web, pois a informação é exibida apenas na "Ferramenta do desenvolvedor" do navegador. Já o `prompt()` e o `alert()` iniciamos uma interação ativa

com o nosso usuário. O `prompt()` mostrará ao usuário uma caixa de diálogo para que insira um dado e a sua programação recebe esse dado. Por exemplo, com este código:

```
var nome = prompt("Insira seu nome");
console.log(nome);
```

Será gerado um resultado assim:

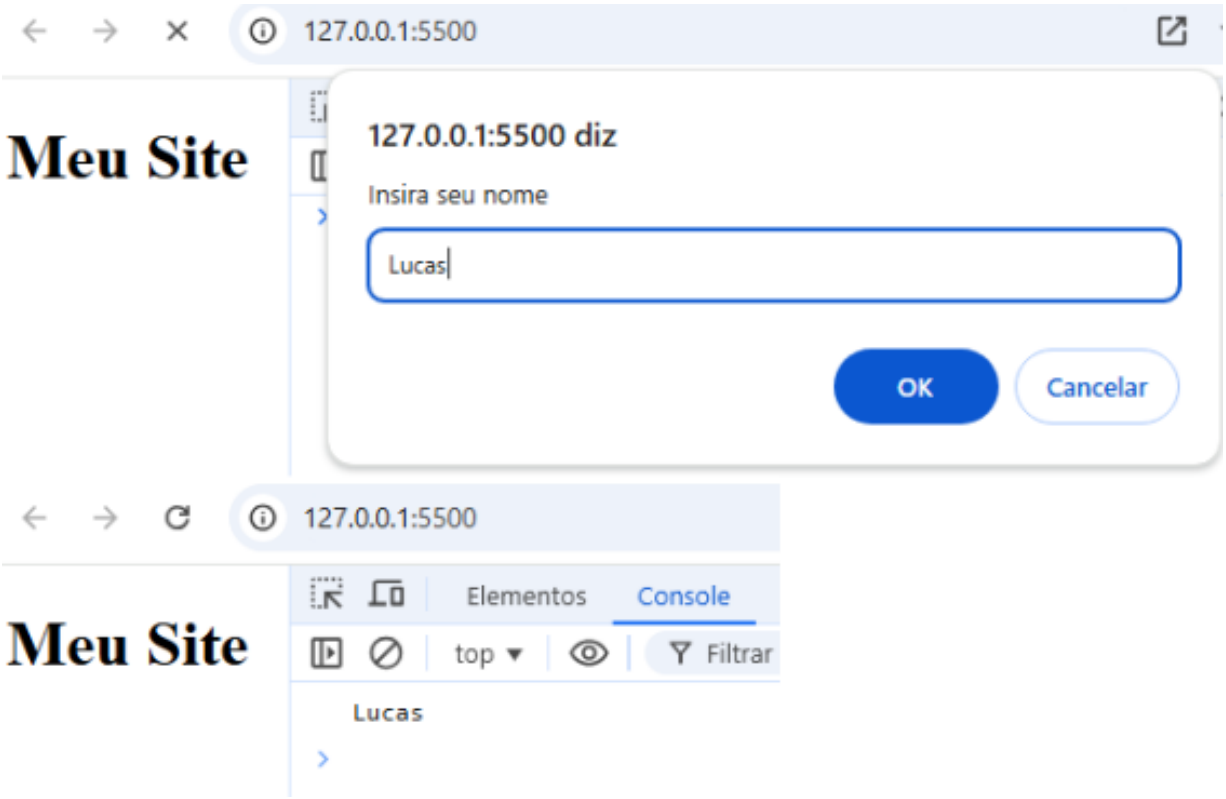


Figura 3.6: Prompt para entrada de informações

Se você executar esse comando verá que o comando prompt abre uma janela e fica aguardando que o usuário digite alguma coisa. Somente depois que ele digita e clica em "OK" a programação continua para a próxima linha. Note que nesta ação o valor que foi digitado pelo usuário é guardado na variável "nome" e depois pode ser usada na sequência do programa. Essa é nossa primeira forma de interagir e obter uma informação fornecida pelo usuário. Chamamos isso de "entrada de dados".

Com esse mesmo código, faça um teste também com o usuário apertando no botão "Cancelar". Você verá no seu console a informação `null`. Isso quer dizer que não foi guardado nada na sua variável. Vale notar que, quando uma variável está nula, é diferente de ela ter o valor 0 ou ela ter o valor " " (um espaço). Valores de espaços ou zeros são diferentes de nulos. Uma analogia que ilustra bem isso é como se você consultasse um saldo bancário e sua conta estivesse com R$ 0,00. Estar com zero reais na conta é diferente de não ter conta no banco.

Já estamos mais habituados com o comando console, então faremos um breve reforço e veremos algumas variações que ele possui. Teste junto:

```
var nome = "Lucas";
console.log("Olá " + nome);

console.warn("Cuidado");
console.error("Deu erro!");
```

Esse código gerará algo como:

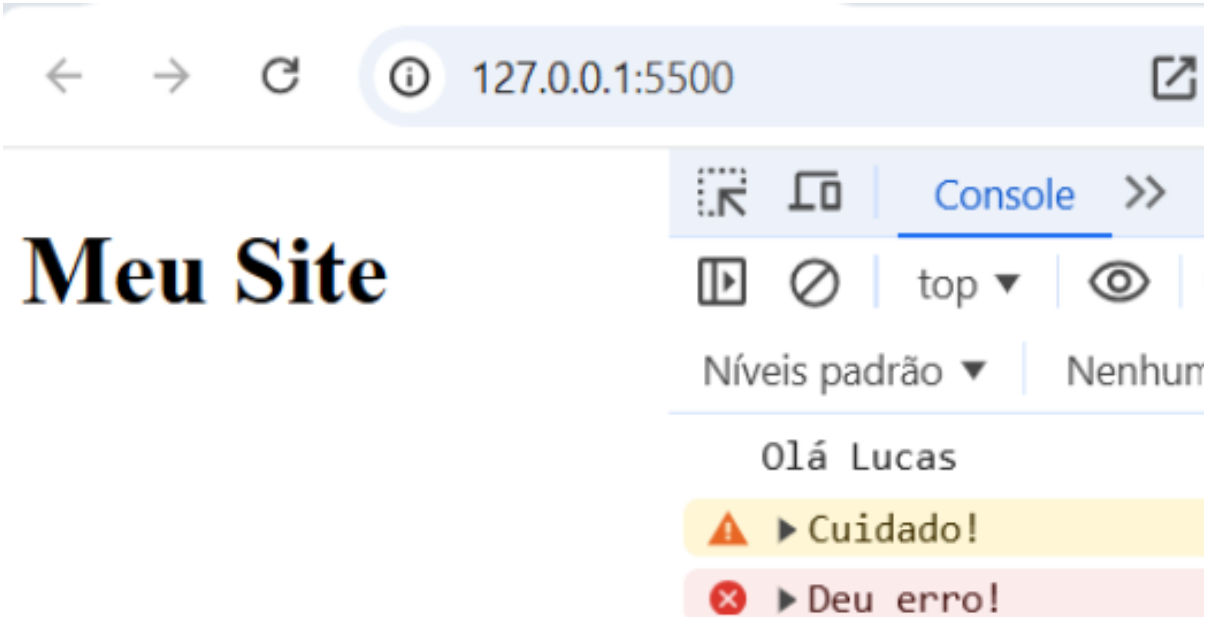


Figura 3.7: Testes no console

Tudo que colocamos dentro dos `( )` chamamos de parâmetros e com o símbolo `+` podemos concatenar (juntar) textos. O `console` tem as variações de `.log()` para exibições comuns; `.warn()` para caracterizar uma atenção; `.error()` para informar um erro. Ainda sendo um comando útil para pessoas que abrem a ferramenta do desenvolvedor, essas variações são úteis para o próprio programador ou para programadores do mesmo projeto.

Por fim, temos o `alert()` que mostra uma janela sobre a página que estivermos acessando, exibindo uma mensagem que for passada como parâmetro, sendo que pode ser uma variável ou não.

```
alert("Olá, desenvolvedor!");
```

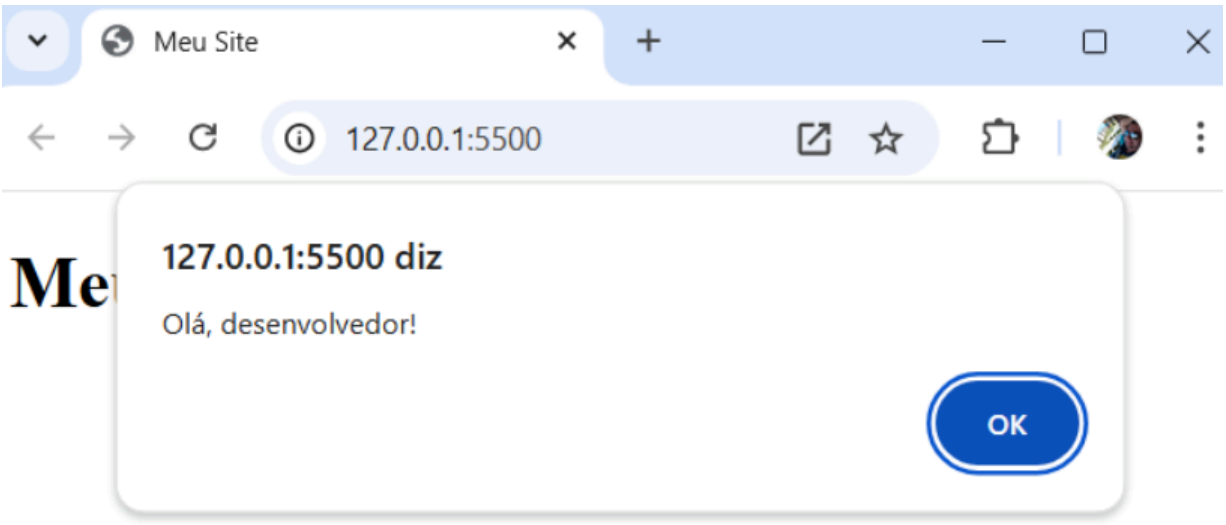


Figura 3.8: Alerta no navegador

Vamos executar um exemplo um pouco mais completo? Confira:

```
var entrada = prompt("Insira seu nome");
var saida = "Olá " + entrada + "! Seja bem-vindo(a) ao meu site";
alert(saida);
```

Observe que temos a entrada de informação no seu sistema, depois concatenamos essa entrada de informação a mais textos que desejamos e atribuímos a outra variável que será nossa saída de informação em um `alert()`.

3.8 Conversão de tipo de dados

Quando usamos o `prompt` para receber alguma informação do usuário, os dados que o navegador recebe são sempre do tipo texto. Vamos aproveitar esse momento para corrigir essa nomenclatura: todo valor textual é chamado de **string**. Ou seja, o `prompt()` retorna uma variável do tipo string. Porém, você pode querer construir um algoritmo que receba uma informação numérica do usuário pelo `prompt`. Verifique na sequência:

```
var x = prompt("Digite um número");
var y = prompt("Digite um número");
var soma = x + y;
var subtracao = x - y;

alert("A soma dos 2 números é: " + soma);
alert("A subtração dos 2 números é: " + subtracao);
```

Ao executar esse código, faça um teste colocando no primeiro `prompt` o valor 10 e no segundo `prompt` o valor 5. A linha 3 realizará uma soma, e a linha 4, uma subtração. Depois teremos 2 alertas que exibirão consecutivamente as mensagens "A soma dos 2 números é: 105" e "A subtração dos 2 números é: 5".

Perceba que a segunda mensagem está correta, pois realmente 10 - 5 = 5, mas a primeira mensagem é um absurdo, pois 10 + 5 não pode ser 105. O que aconteceu?

Por padrão o `prompt` recebe os dados inseridos pelo usuário como uma string. Assim, mesmo que o usuário digite um número, como 10, o valor será interpretado como texto e armazenado dessa forma na variável. Em seguida, ao realizar uma operação com o símbolo `+`, o programa pode interpretar essa ação como uma concatenação de textos — e não como uma soma numérica — já que o operador `+` serve para ambas as finalidades. Por outro lado, o operador é exclusivo para subtrações com valores numéricos, o que evita esse tipo de ambiguidade. Portanto, ao utilizar entradas numéricas com

`prompt`, é fundamental indicar ao programa o tipo de dado que se deseja manipular. Veja, a seguir, como esse mesmo código pode ser ajustado com conversão explícita de tipos:

```
var x = prompt("Digite um número");
var y = prompt("Digite um número");

x = parseInt(x);
y = parseInt(y);

var soma = x + y;
var subtracao = x - y;

alert("A soma dos 2 números é: " + soma);
alert("A subtração dos 2 números é: " + subtracao);
```

Neste código, as linhas 4 e 5 indicam explicitamente que os valores armazenados em `x` e `y` serão convertidos para o tipo inteiro. O `parseInt()` realiza essa conversão de uma string para um número inteiro, ou seja, `x` passa a conter o valor numérico equivalente ao que estava originalmente em formato de texto. O mesmo ocorre com `y`. O termo `Int`, no nome da função, vem justamente de *"integer"*, que significa inteiro, em inglês. Caso você precise converter uma string para um número com casas decimais, pode utilizar o comando `parseFloat()`. E se quiser fazer o caminho inverso — transformar um número em texto —, uma maneira simples de fazer isso é somar o número com uma string vazia (`""`), o que força a conversão para o tipo textual.

```
var x = 10;
var y = "10";

var resultado = x + parseInt(y);
alert(resultado);

resultado = x + y + "";
alert(resultado);
```

Analisando esse código:

- linha 1: declaração de variável `x`, do tipo numérico inteiro;
- linha 2: declaração de variável `y` string. O programa sabe que é string devido às `""` envolvendo o 10;
- linha 4: declaração de uma variável resultado que recebe a soma de `x` e `y`. Para que fique claro para o programa que você deseja somar números, existe uma conversão da variável `y`. Essa conversão somente é necessária para a variável `y`, pois ela é a única declarada como string (envolvida por aspas).
- linha 5: exibe a variável `resultado`;
- linha 7: outro cálculo é atribuído a variável `resultado`, porém no cálculo tem a soma de um texto vazio `""`. Somente o fato de adicionar esse texto vazio faz com que o JavaScript converta todo o cálculo em texto. A partir daí, ele não executa uma conta, e sim uma junção de texto;
- linha 8: exibe o valor da variável `resultado` que agora tem um texto em seu valor. A saída será `1010`.

3.9 Controlando o fluxo do seu algoritmo

Você deve ter percebido que o JavaScript faz a execução do seu código de forma sequencial, linha por linha, de cima para baixo, em um fluxo reto, uma sentença após a outra.

Figura 3.9: Fluxo sequencial de algoritmo

Vimos que isso se encaixa na definição de algoritmo: uma sequência de passos finitos para executar determinada tarefa. Seja uma tarefa simples de soma ou uma tarefa de geração de cadastro completo com emissão de relatórios ou gráficos.

Imagine um algoritmo que verifica a altura de uma pessoa e, se for menor que 1,40 metros, não poderá entrar no parque de diversões. Perceba que há uma regra neste algoritmo e, se acontecer alguma coisa, o algoritmo executa uma ação, senão ele executa outra ação, havendo dois caminhos possíveis:

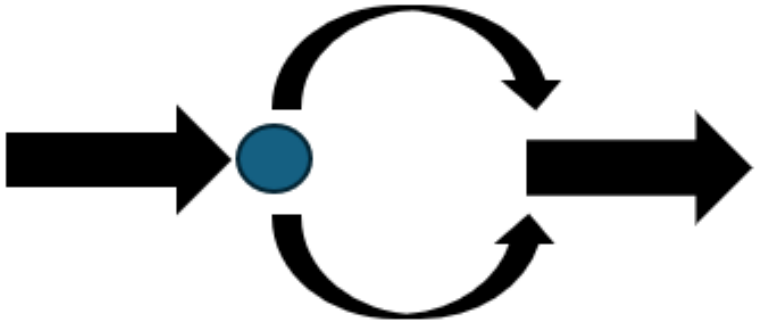


Figura 3.10: Fluxo condicional de algoritmo

SE pessoa for menor que 1,40 –> Bloqueia a entrada

SENÃO, permite a entrada

Assim, dependendo da altura da pessoa, serão executados determinados trechos de código; caso contrário, outros. Por isso, vamos aprender a controlar o fluxo de execução do nosso algoritmo.

Condicionais

Os comandos condicionais são aqueles que permitem decidir se determinado trecho de comandos deve ser executado ou não, a partir do resultado de uma expressão lógica ou relacional. Funciona assim:

SE (condição verdadeira)

ENTÃO executa esses códigos

SENÃO executa esses outros códigos

Para gerar esses controles, vamos começar a explorar os valores `booleanos`. Eles são valores que assumem apenas duas possibilidades: true ou false (verdadeiro ou falso). Assim, para que nosso programa tome um caminho ou outro, vamos efetuar alguma comparação para obter um resultado `true` ou `false`. Essa comparação é feita pela **estrutura if**.

Observe esses dois códigos que se diferenciam apenas pelo valor de entrada da variável:

```
var altura = 1.33;
if(altura < 1.40){
    alert("Entrada bloqueada");
}
else{
    alert("Entrada permitida.");
}
```

```
var altura = 1.83;
if(altura < 1.40){
    alert("Entrada bloqueada");
}
else{
    alert("Entrada permitida.");
}
```

Respectivamente os resultados serão:

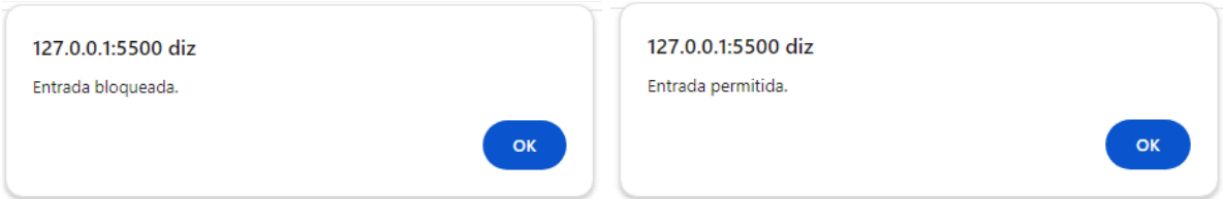


Figura 3.11: Resultado do alerta

Dependendo do valor atribuído à variável altura, o mesmo código pode gerar resultados diferentes. Na linha 3, é feita uma comparação entre o valor da variável `altura` e o número 1.40. O símbolo `<` verifica se um valor é **menor** do que o outro. Quando

`altura = 1.33`, essa comparação resulta em `true`, pois 1.33 é, de fato, menor que 1.40. Sendo assim, o bloco dentro do `if` é executado. Já no segundo exemplo, em que altura = 1.83, a comparação retorna `false`, e o programa segue para o bloco `else`, executando o código correspondente.

Esse tipo de estrutura condicional pode ser interpretado, de forma comparativa com a linguagem natural, da seguinte maneira:

```
SE (isso aqui é verdade) ENTÃO  ->  if(true){
        executa isso aqui       ->      alert("Uma verdade!");
SENÃO                           ->    } else {
        executa esse outro aqui ->      alert("Um falso");
FIM                             ->  }
```

Agora também podemos comentar sobre os `{ }`. Eles são delimitadores de início e fim. Isso quer dizer que na estrutura `if` (ou em qualquer outra) pode haver mais códigos dentro da condição quando for `true` ou mais códigos quando for `false`. Tudo dentro das chaves é executado. Vamos usar em diversos lugares na programação JavaScript e eles sempre são usados para delimitar trechos de código sendo sempre `{` o início e `}` o término.

Preciso reforçar que nesta estrutura `if` tem dois caminhos possíveis, mas o `if` pode aparecer sem estar acompanhado do `else`. Às vezes, a verificação é unicamente se apenas uma coisa acontece e assim não haverá um SENÃO.

```
if(numero_sorteado == 10){
    alert("Você ganhou o prêmio.");
}
```

Temos um exemplo de alguma coisa que vai acontecer e, se não acontecer, nada mais precisará ser feito. Uma única condição.

3.10 Operadores relacionais

Agora que compreendemos como funciona uma condicional, vamos explorar as diferentes maneiras de realizar comparações. Em uma situação anterior, comparamos se uma variável era **menor que** um determinado valor. No entanto, há diversas outras possibilidades de comparação, cada uma com um símbolo específico. A seguir, vamos conhecer esses operadores e compreender como eles funcionam na prática:

Operador	Significado
`==`	Igual a
`===`	Valor e tipo igual a
`!=`	Diferente de
`>`	Maior que
`<`	Menor que
`>=`	Maior ou igual a
`<=`	Menor ou igual a

Vou deixar algumas outras comparações de exemplo caso queira executar:

```
var preco = 9.99;

//Verificando se algo é diferente
if(preco != 0){
    alert("O produto não é gratis.");
}

//Verificando se o preço é maior que 5
if(preco > 5){
    alert("O preço é maior que 5.");
}

//Verificando se o preço é igual
```

```
if(preco == 9.99){
    alert("O preço deste produto é 9.99.");
}
```

Agora, com vários exemplos e com a tabela dos operadores que vimos anteriormente, é possível que você faça alguns outros treinos e testes sozinho.

3.11 Operadores lógicos

Operadores lógicos são utilizados para combinar resultados de expressões. Com eles é possível fazer várias comparações de uma única vez retornando um único resultado final de verdadeiro e falso.

Ainda no exemplo de parque de diversões, vamos imaginar que para entrar nos brinquedos é necessário ter mais de 1.40 metros de altura e também mais de 16 anos. Observe que temos uma nova condição que precisa ser analisada ao mesmo tempo. Agora existem 2 requisitos mínimos para entrar no parque de diversões e se apenas um deles for verdadeiro ainda não será suficiente para permitir a entrada. Nesses casos, usamos operadores lógicos para validar mais de uma condição simultaneamente.

São três operadores lógicos que nos ajudam a validar mais de uma mesma opção em estruturas condicionais:

- Operador `AND` representado pelos símbolos `&&` : Esse operador necessita que **todas** as comparações na expressão sejam verdadeiras para resultar em uma validação verdadeira e assim acessar a estrutura. A saber:

Linguagem natural

```
SE (altura maior que 1.40 AND idade maior que 16 )
    Permite entrada
```

Código JS

```
if(altura > 1.40 && idade > 16){
    alert("Entrada Liberada.");
}
```

No exemplo, o código JavaScript acessará o alerta de "Entrada Liberada" se as duas condições forem atendidas. Ou seja, o valor da variável `altura` tem que ser maior que 1.40 e a variável `idade` tem que ser maior que 16.

- Operador `OR` representado pelos símbolos `||` : Esse operador necessita que **apenas** uma comparação na expressão seja verdadeira para resultar em uma validação verdadeira e assim acessar a estrutura. A saber:

Linguagem natural

```
SE (altura maior que 1.40 OR idade maior que 16 )
    Permite entrada
```

Código JS

```
if(altura > 1.40 || idade > 16){
    alert("Entrada Liberada.");
}
```

O código JavaScript acessará o alerta de "Entrada Liberada" se apenas uma das condições for atendida. Ou seja, se a variável `altura` for maior que 1.40 `OU` se a variável `idade` for maior que 16. A diferença ao mudar o operador é bem grande e nos dá uma noção de que a quantidade de possibilidades é muito vasta.

- Operador `NOT` representado pelo símbolo `!` : Esse operador tem a finalidade de inverter o resultado da comparação. Ele pode ser adicionado em uma ou mais comparações. O exemplo dele será um pouco diferente. Imagine que você quer emitir o alerta apenas para quem **não** poderá entrar no parque de diversões e apenas para quem tem menos de 16 anos:

Linguagem natural

```
SE (NOT idade menor que 16 )
    Entrada bloqueada
```

Código JS

```
if(!idade < 16){
    alert("Entrada bloqueada.");
}
```

O código JavaScript verifica se variável `idade` é menor que 16. A primeira coisa feita é essa comparação e se ela for verdadeira o operador `!` inverte o resultado e torna ela falsa e vice-versa. Ou seja, se a variável `idade` estiver com valor 17 o `if` vai comparar 17 com 16 e encontrará um resultado falso; porém, o operador `NOT` inverte esse resultado para `true` e `true` acessa a condição, entrando no `alert` .

Código	Operador	Tradução	Significado
&&	AND	E	Adição
|	OR	OU	Contraste
!	NOT	NÃO	Negação

Para entender melhor como esses operadores funcionam na prática, podemos observar a chamada **Tabela Verdade**, que mostra os possíveis resultados das combinações entre condições:

- Operador AND (`&&`):

1ª Condição	2ª Condição	Resultado
V	V	**V**

1ª Condição	2ª Condição	Resultado
V	F	**F**
F	V	**F**
F	F	**F**

- Operador OR (`||`):

1ª Condição	2ª Condição	Resultado
V	V	**V**
V	F	**V**
F	V	**V**
F	F	**F**

- Operador NOT (`!`):

Condição	Resultado
V	**F**
F	**V**

Suponha que temos as variáveis `X = 4`, `Y = 7` e `Z = 2` os resultados dessas expressões seriam:

Expressões	Resultado
X == Y → (F) **AND** Y >= Z → V	**F**
X != Y → (V) **OR** Y <= Z → F	**V**
X > Y → (F) **NOT**	**V**

Vamos ver alguns exemplos com o código JavaScript. Diante de uma combinação de operadores `&&` (AND), sendo necessários que todas as comparações sejam verdadeiras para que a condição composta seja verdadeira:

```
var numero1 = prompt("Insira um numero");
var numero2 = prompt("Insira outro numero");
if((numero1 != "") && (numero2 != "")) {
    var resultado = numero1 * numero2;
    alert("A multiplicação dos numeros digitados é " + resultado);
}
else{
    alert("Erro! Não é possível multiplicar se um dos números for
vazio.");
}
```

Analisando esse código, temos duas condições que verificam se os números digitados pelo usuário nos prompts são diferentes de vazio e a multiplicação do `numero1` com o `numero2` SOMENTE será feita se os dois números digitados forem diferentes de vazio (seria o caso do usuário digitar OK sem colocar nada no campo do prompt). Ou seja, graças ao operador `&&` , as duas condições têm que ser verdadeiras.

```
var nome1 = prompt("Insira um nome");
var nome2 = prompt("Insira outro nome");
if((nome1 != "") || (nome2 != "")) {
    alert("Primeiro nome digitado: " + nome1);
    alert("Segundo nome digitado: " + nome2);
}
else{
```

```
    alert("Nenhum nome foi digitado.");
}
```

Nesse exemplo, se apenas um dos nomes for digitado, aparecerão 2 alertas mostrando os nomes e, caso um seja vazio, um dos alertas mostrará um nome vazio. Embora não seja o ideal, não teremos maiores problemas no código, diferente do código anterior onde a multiplicação de qualquer valor vazio gera erro no navegador. Outro caso seria se o usuário não digitasse em nenhum dos 2 prompts, então a mensagem ideal seria a do "Nenhum nome foi digitado.". Ou seja, o operador `||` traz essa liberdade de que apenas uma das condições seja atendida.

3.12 É possível combinar operadores relacionais e lógicos?

Isso é muito comum. Na verdade, esse tópico é só para sanar qualquer dúvida que ocasionalmente exista quanto ao uso dos operadores. Observe novamente nos exemplos anteriores que, com os operadores relacionais, fazemos comparações entre valores, e os operadores lógicos usamos somente quando temos que fazer mais de uma comparação ao mesmo tempo. Isso faz com que, quase obrigatoriamente, os operadores lógicos sejam usados para complementar comparações feitas pelos operadores relacionais. Então, quando precisar de uma estrutura `if` mais robusta, com certeza haverá combinação entre os operadores relacionais e lógicos. Vamos fazer um programa de exemplo?

```
var idade = prompt("Digite sua idade");
if(idade < 14){
    alert("Ainda criança");
}
else if(idade < 17){
    alert("Está jovem");
}
```

```
else if( (idade >=18) && (idade < 60) ) {
    alert("É adulto!");
}
else{
    alert("É idoso!");
}
```

Com isso, consolidamos uma base introdutória de JavaScript e lógica de programação. Mas por que parar no básico? Afinal, queremos gerar Dashboards informativos e atraentes. Vamos nos aprofundar.

CAPÍTULO 4
Funções e estruturas de repetição

4.1 Funções

Quando estamos programando, é muito comum usarmos o mesmo algoritmo mais de uma vez. Uma função é um conjunto de instruções agrupadas para realizar uma tarefa, reutilizável em diferentes partes do código.

As funções trazem diversas vantagens: evitam a duplicidade de código, garantem mais organização e clareza ao código, permitem dividir tarefas complexas em partes menores e mais simples, melhoram a legibilidade do código, além de fornecer facilidade e rapidez para fazer modificações ou manutenções.

Assim como as variáveis, as funções devem ser declaradas, porém com suas próprias características. Isso é feito através da palavra reservada `function`, seguida de um nome (que geralmente indica o que aquele conjunto de comandos executa), os parênteses `()` e, por fim, as chaves `{}`, que são utilizadas como delimitadores de início e fim da função.

```
function BoasVindas() { //Declaração, nome, parênteses e delimitador inicial da função
    alert("Olá! Seja bem-vindo ao site!"); //Conteúdo da função
} //delimitador final da função
```

A função foi declarada, e isso pode ser feito em qualquer parte do seu código. Seu programa entenderá que existe uma função, mas não a executará até que você faça uma **chamada da função**. Nós podemos usar essa função em qualquer momento e quantas vezes forem necessárias. Sempre que a função for chamada pelo nome, ela executará todos os comandos que foram programados entre as suas chaves.

No seu arquivo de script faça o seguinte teste:

```
function BoasVindas() {
    var nome = prompt("Digite seu nome:");
    alert("Olá, " + nome + "! Seja bem-vindo!");
}

BoasVindas(); //Chama a função para ser executada
BoasVindas(); //Chama a função para ser executada novamente
BoasVindas(); //Chama a função para ser executada novamente
```

Você notará que o pedido para digitar um nome, assim como as boas-vindas para esse nome, aparecerá três vezes na sua tela, pois neste código chamamos a função três vezes. A mesma função deu boas-vindas a três pessoas diferentes — e esse é o início do poder das funções. Um único bloco de código, bem organizado, nos permite economizar várias outras linhas de programação.

As funções simples não precisam de informações para funcionar, mas quando começamos a codificar funções mais complexas nos deparamos com a necessidade de receber informações.

4.2 Funções com parâmetros

Essa função de boas-vindas, assim como diversas outras, pode não precisar de nenhuma informação para funcionar. No entanto, à medida que aumentamos a complexidade, temos a opção de parametrizar alguns elementos e enviar valores para elas. Fazemos isso por meio dos **argumentos da função**. Para que a função receba esses argumentos, ao chamá-la, passamos esses valores pelos **parâmetros da função**, entre parênteses, logo após o nome. Assim:

```
function Soma(a, b){
    resultado = a + b;
    console.log("Resultado da soma: " + resultado);
}

Soma(10, 5);
Soma(20, 20);
```

A função acima tem, entre seus parênteses, as variáveis `a` e `b`. Elas foram declaradas dentro dos `()` da função e, portanto, só podem ser dentro dela. Como essa função possui duas variáveis na sua declaração, sempre que for chamada, precisa receber dois valores, como no exemplo.

Isso nos dá muitas opções ao criar uma função. Podemos pensar em códigos que serão muito usados e às vezes só precisamos de valores diferentes. Assim, aproveitamos a mesma função passando apenas valores diferentes. Nesse exemplo, a função é responsável por somar dois números e exibir o resultado no console. Como ela foi chamada duas vezes com valores diferentes nos seus parâmetros, o resultado no console foi "Resultado da soma: 10" e "Resultado da soma: 40".

O uso de argumentos e parâmetros não termina aqui. Nesse código, usamos apenas dois argumentos, mas não há limite para a quantidade de argumentos que uma função pode receber. Você pode criar uma função com quantos argumentos forem necessários e, sempre que chamá-la, deverá fornecer essa mesma quantidade de informações. Isso significa que, se uma função tiver cinco parâmetros, quando for chamada precisará ter cinco informações. E como o JavaScript sabe qual informação pertence a qual argumento? Pela ordem em que eles são passados. Ou seja:

- Vamos considerar que em nosso programa existe a função `CalculaMedia(x, y, z)` e na hora de chamar fazemos:

```
CalculaMedia(5, 7, 9); //Nessa chamada: x = 5, y = 7, z = 9
CalculaMedia(10, 20, 30); //Nessa chamada: x = 10, y = 20, z = 30
```

```
CalculaMedia(8, 2, 4); //Nessa chamada: x = 8, y = 2, z = 4
```

Simples assim. Não precisamos nem saber o conteúdo da função. Os valores serão sempre usados na ordem em que **foram enviados**.

E se eu passar a quantidade errada de argumentos?

Em JavaScript, se você passar menos argumentos do que os parâmetros definidos, os parâmetros ausentes serão considerados `undefined` (indefinidos). Se passar mais argumentos, os extras serão simplesmente ignorados. Confira:

```
function Soma(a, b){
    console.log("a: " + a);
    console.log("b: " + b);
    console.log("Resultado da soma: ");
    console.log(a + b);
}

Soma(5, 10, 20); // 20 será ignorado → Resultado: 15
Soma(5); //b será undefined → Resultado será um NaN, que é um tratamento do JS que significa "Not a Number".
```

Por isso, é sempre importante garantir que você está passando a quantidade correta de argumentos ou que sua função esteja preparada para lidar com variações, quando isso fizer sentido.

O que é o return nas funções?

As funções geralmente são utilizadas para executar determinada ação específica colocando algum valor na tela ou alterando alguma informação na página. Elas executam quando são chamadas e o código continua; ou então elas podem ser chamadas para encontrar determinado valor antes que o código prossiga em seu fluxo. Vamos analisar a função a seguir:

```
function Media(a, b){
    var soma = a + b;
```

```
    return soma / 2;
}
```

Essa função tem em sua última linha a palavra `return`, que faz exatamente o que o nome sugere: ela retorna um valor para quem a chamou. Vejamos um programa que usa a função `Media()`:

```
var nota1 = 10;
var nota2 = 8;

var resultado = Media(nota1, nota2);
console.log(resultado);
```

No momento em que `resultado` é declarado, ocorre uma chamada para a função. Nesse instante, a função é executada, recebendo `nota1`, que corresponde ao valor de `a` (neste caso, 10), e a `nota2`, que corresponde ao valor de `b` (neste caso, 8). É realizada uma soma e, em seguida, uma divisão por 2, cujo resultado é retornado.

O comando `return` funciona como uma ponte: ele envia o resultado da função de volta para quem a chamou. Tudo o que foi processado na função `Media` é retornado no momento da chamada, e o valor calculado é armazenado em `resultado`.

4.3 Função anônima

Uma função pode ser definida sem nome e utilizada para ser atribuída a uma variável. Isso mesmo. Variáveis podem receber funções inteiras sem nome. Essa variável se torna o identificador desta função sem nome:

```
//função sem nome com dois parâmetros
var resultado_soma = function (i, j) { return i + j }

//Exibição da função
console.log(resultado_soma(10, 20));
```

Então veja que podemos criar uma função sem nome e armazená-la dentro de uma variável. A partir disso, usamos o nome da variável para "chamar" a função, como se ela fosse uma função com nome. Na prática, o nome da variável passa a ser a referência dessa função. Por isso, mesmo sendo chamada de "função anônima", ela acaba sendo acessada normalmente, como qualquer outra.

Esse tipo de função é muito útil em situações rápidas, como pequenos cálculos ou ações pontuais dentro de outra função. Se você ainda não se sentir confortável usando funções anônimas, não se preocupe — é normal. Elas aparecem mais à frente em contextos como funções de retorno e Programação Funcional. O mais importante é saber que elas existem e que você já deu o primeiro passo para entendê-las.

4.4 Função de seta

É uma forma ainda mais abreviada de usar as funções anônimas. Também podem ser chamadas de **Arrow Function**. Veja o exemplo anterior sendo usado com as funções de seta:

```
//função sem nome com 2 parâmetros
var resultado_soma = (i, j) => { return i + j }

//Exibição da função
console.log(resultado_soma(10, 20));
```

4.5 Loops e laços de repetição

Loops, laços de repetição ou ciclos de iteração são formas de executar algo **várias vezes**. Essa necessidade é muito comum em qualquer programa, afinal, uma das maiores vantagens de qualquer

programa é automatizar tarefas que precisam ser feitas repetidamente.

O JavaScript oferece três estruturas principais para criar repetições: `for`, `while` e `do-while`. Elas permitem que um bloco de código seja executado várias vezes, de forma automática. Essas estruturas podem ser agrupadas em dois tipos de loops, dependendo da lógica de repetição:

- Loops por contagem: repetem o código um número determinado de vezes, normalmente usando a estrutura `for`.
- Loops condicionais: repetem o código enquanto uma condição for verdadeira, utilizando `while` ou `do-while`.

4.6 Estrutura for

Essa estrutura tem a seguinte sintaxe:

```
for(Começo; Condição de Parada; Atualização)
{ //chave para delimitar o início da repetição
    //Todos os códigos aqui entre {} serão executados enquanto o loop durar
} //chave para delimitar o término da repetição
```

No `for`, usamos uma variável de controle para acompanhar o andamento da repetição:

- Em `Começo`, definimos o valor inicial dessa variável de controle.
- A `Condição de Parada` é uma expressão lógica que será avaliada antes de cada repetição; se for verdadeira, o bloco entre as chaves será executado novamente.
- Em `Atualização`, indicamos o que acontecerá com a variável a cada ciclo (por exemplo, incrementar ou decrementar seu valor).

Vamos conferir um exemplo simples:

```
for(var i=0; i < 10; i++)
{
    console.log("Valor de i é: " + i);
}
```

Tentando traduzir essa estrutura de repetição para o português, seria algo como: " `i` começa em zero, enquanto `i` for menor que 10, acrescenta `1` ". Esse código produzirá o resultado:

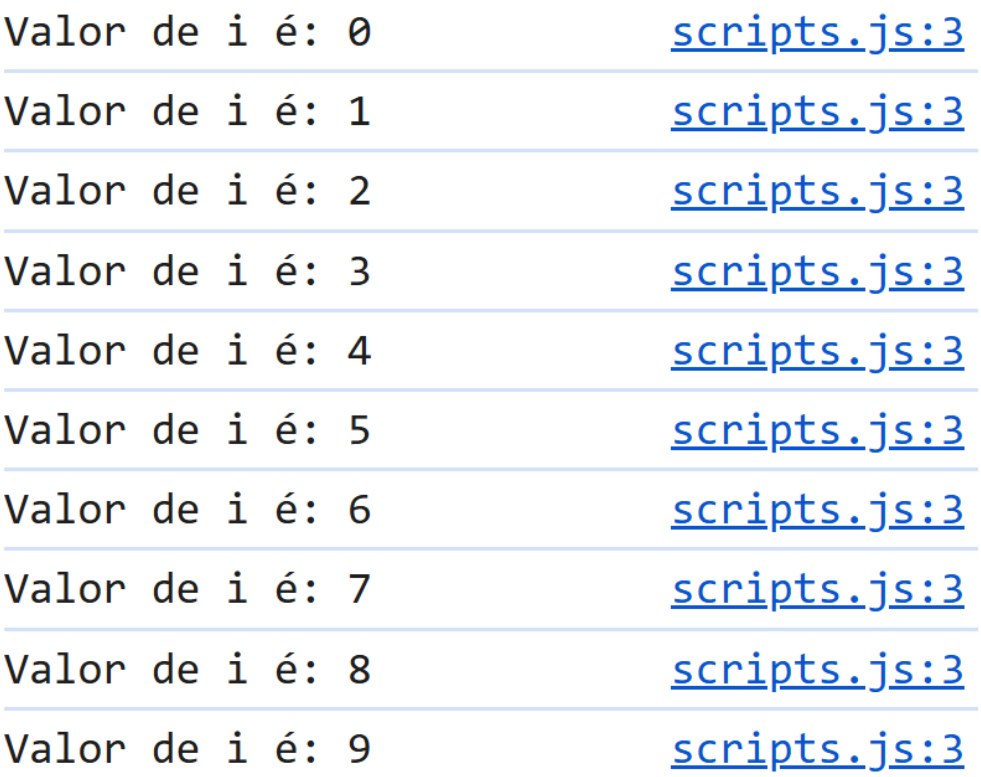


Figura 4.1: Resultado de um loop com for

O código dentro das `{}` do `for` foi executado conforme as regras que definimos. Como o JavaScript leu esse código:

- Cheguei a um `for` , então vou fazer uma repetição;
- Essa repetição vai ser controlada por `i` que vale 0;
- Vou parar quando `i` for menor que 10 e agora `i` é 0;
- Então executo tudo dentro de `{}` e somo 1 ao i;
- Vou parar quando `i` for menor que 10 e agora `i` é 1;
- Então executo tudo dentro de `{}` e somo 1 ao i;

- Vou parar quando `i` for menor que 10 e agora `i` é 2;
- Então executo tudo dentro de `{}` e somo 1 ao i;
- Vou parar quando `i` for menor que 10 e agora `i` é 3;
- Então executo tudo dentro de `{}` e somo 1 ao `i`;
- ... (Isso se repete algumas vezes) ...
- Vou parar quando `i` for menor que 10 e agora `i` é 9;
- Então executo tudo dentro de `{}` e somo 1 ao `i`;
- Vou parar quando `i` for menor que 10 e agora `i` é 10;
- Opa! 10 não é menor que 10! Atingi a condição de `i < 10`;
- Então parei a execução.

Legal, né?! Um exemplo clássico de algoritmo com `for` quando estamos aprendendo a programar é fazer uma tabuada. Em vez de você fazer:

```
let tabuada = parseInt(prompt("Insira o número da tabuada:"));
console.log(tabuada + " X 1 = " + tabuada * 1);
console.log(tabuada + " X 2 = " + tabuada * 2);
console.log(tabuada + " X 3 = " + tabuada * 3);
console.log(tabuada + " X 4 = " + tabuada * 4);
console.log(tabuada + " X 5 = " + tabuada * 5);
console.log(tabuada + " X 6 = " + tabuada * 6);
console.log(tabuada + " X 7 = " + tabuada * 7);
console.log(tabuada + " X 8 = " + tabuada * 8);
console.log(tabuada + " X 9 = " + tabuada * 9);
console.log(tabuada + " X 10 = " + tabuada * 10);
```

Você pode simplesmente fazer:

```
for (var i = 1; i <= 10; i++) {
    console.log(tabuada + " X " + i + " = " + tabuada * i);
}
```

Viu como conseguimos economizar código e repetir diversas vezes? Esse exemplo mostra uma tabuada até 10. Mas e se quiséssemos uma que fosse até 1 milhão? Seria bem chato fazer um código com 1 milhão de linhas repetidas mudando somente o número a ser multiplicado. Já com o `for` só precisamos atualizar o código anterior para:

```
for (var i = 1; i <= 1000000; i++) {
    console.log(tabuada + " X " + i + " = " + tabuada * i);
}
```

4.7 Estrutura while

Essa estrutura também serve para gerar repetições. Nela, só precisamos definir uma condição de parada, veja:

```
var condicao;
while(condicao != 0){
    alert("Estou dentro de um loop!");

    //Na próxima repetição será verificado se a variável recebeu o valor 0.
    //Caso contrário, repetirá novamente.
    condicao = prompt("Digite 0 para para de repetir!");
}
```

Mas tenha cuidado! Se essa condição não for devidamente planejada, você pode gerar o chamado "loop infinito", que é quando a condição nunca encontra uma parada. Por exemplo:

```
var condicao = true;
while(condicao == true){
    alert("Estou dentro de um loop!");
}
```

Nesse código, não há nada dentro do `while` que altere o valor da variável `condicao`, mantendo-a sempre verdadeira e fazendo com que a condição de parada nunca aconteça. Assim, seu código vai rodar infinitamente até travar seu navegador ou até ser necessário fechar a aba.

4.8 Estrutura do-while

Essa estrutura tem um funcionamento muito semelhante ao `while`, mas com uma diferença muito significativa: **ela sempre será executada pelo menos uma vez**.

Enquanto a estrutura `while` verifica a condição antes de executar o bloco de código, a estrutura `do-while` executa primeiro e só depois verifica se deve continuar repetindo.

```
var valor = 0;

while(valor == 1){
    //Verifica a condição e depois executa. Neste caso, a condição é falsa
então não executa.
    console.log("Estou dentro do while!");
}

do { //Executa uma vez, mesmo com a condição falsa.
    console.log("Estou dentro do loop do-while!");
} while(valor == 1); //Aqui a condição é falsa, mas o bloco já foi
executado uma vez.
```

Acima temos um mesmo *loop*, porém com as estruturas diferentes. Se você rodar esse código, vai perceber que somente o segundo *loop* vai executar porque nos dois casos a condição é falsa. Mas, como vimos, o `do-while` executa antes de verificar a condição. Ao usá-lo, sempre teremos pelo menos uma execução do conteúdo do *loop*.

4.9 Switch

É uma estrutura projetada para administrar várias condições sobre a mesma variável. Tecnicamente, isso já pode ser feito com o `if`, mas

em alguns casos o `switch` é mais organizado. Vamos ver uma comparação dessas estruturas.

```
console.log("Selecione uma das opções a seguir:");
console.log("1 - sorvete de morango");
console.log("2 - sorvete de chocolate");
console.log("3 - sorvete de baunilha");
console.log("4 - sorvete de flocos");

var opcao = prompt("Digite aqui o sabor desejado:");
switch(opcao){
    case "1":
        console.log("Você escolheu sorvete de morango");
        break;
    case "2":
        console.log("Você escolheu sorvete de chocolate");
        break;
    case "3":
        console.log("Você escolheu sorvete de baunilha");
        break;
    case "4":
        console.log("Você escolheu sorvete de flocos");
        break;
    default:
        console.log("Opção inválida");
        break;
}
```

Dependendo do que for digitado no `prompt` , será exibida uma das opções no console. Esse mesmo algoritmo pode ser criado com o `if/else` .

```
console.log("Selecione uma das opções a seguir:");
console.log("1 - sorvete de morango");
console.log("2 - sorvete de chocolate");
console.log("3 - sorvete de baunilha");
console.log("4 - sorvete de flocos");
var opcao = prompt("Digite aqui o sabor desejado:");
if(opcao == "1"){
    console.log("Você escolheu sorvete de morango");
```

```
}
else if(opcao == "2"){
    console.log("Você escolheu sorvete de chocolate");
}
else if(opcao == "3"){
    console.log("Você escolheu sorvete de baunilha");
}
else if(opcao == "4"){
    console.log("Você escolheu sorvete de flocos");
}
else{
    console.log("Opção inválida");
}
```

O `if/else` é mais indicado quando vamos fazer comparações entre as coisas. Já a estrutura `switch` tem a ideia de oferecer algumas opções dentre as quais uma deve ser escolhida. A característica do `switch` sempre trará uma comparação única: se algo é igual a alguma coisa; ou se a opção é essa. Já o `if/else` é para comparações: se isso está entre alguma coisa; ou se outra coisa é maior que isso. Podemos definir como: use `switch` para escolhas simples e `if/else` para escolhas complexas.

Fizemos o mesmo algoritmo usando duas estruturas diferentes. Tem alguma diferença? Qual deles é melhor usar? Isso vai depender da comparação que deseja fazer. No caso da escolha de um sorvete, vamos imaginar que temos 100 opções de sorvete. Com a estrutura de simples escolha do `switch`, a opção digitada será direcionada diretamente à opção correta. Já com a estrutura `if/else`, que é um comparador mais complexo, ele fará a comparação até encontrar a opção correta. Então, se imaginarmos que a pessoa escolheu o sorvete com a opção 100, o `if/else` só encontrará a opção correta na centésima comparação. Portanto, **nesse caso**, o uso do `if/else` pode resultar em maior consumo de processamento.

Após aprendermos sobre funções e estruturas de repetição, é o momento de nos aprofundarmos nas estruturas de dados que são amplamente utilizadas em aplicações JavaScript. Vamos entender

como trabalhar com objetos, classes, arrays e outras formas de armazenar e manipular dados.

Capítulo 5
Trabalhando com dados em JavaScript

A lógica de um programa não existe no vazio — ela trabalha em cima de dados. No JavaScript, temos diversas formas de representar e manipular essas informações. Neste capítulo, vamos aprender como criar e lidar com estruturas fundamentais como objetos, classes e arrays, além de explorar alguns recursos nativos extremamente úteis para cálculos e datas.

5.1 Objetos

No JavaScript, e também em qualquer outra linguagem de programação, os objetos podem ser comparados com objetos da vida real. Qualquer objeto possui características ou propriedades. Assim como uma caneta possui uma marca, uma cor, tamanho, peso, no JS os objetos têm propriedades que definem suas características. Eles são definidos entre chaves `{}` e os valores são listados com vírgula para separar as propriedades. Por exemplo:

```
var caneta = {
    cor: 'azul',
    preco: 3.9,
    marca: 'bic',
    peso_gr: 10
}

console.log(caneta);
```

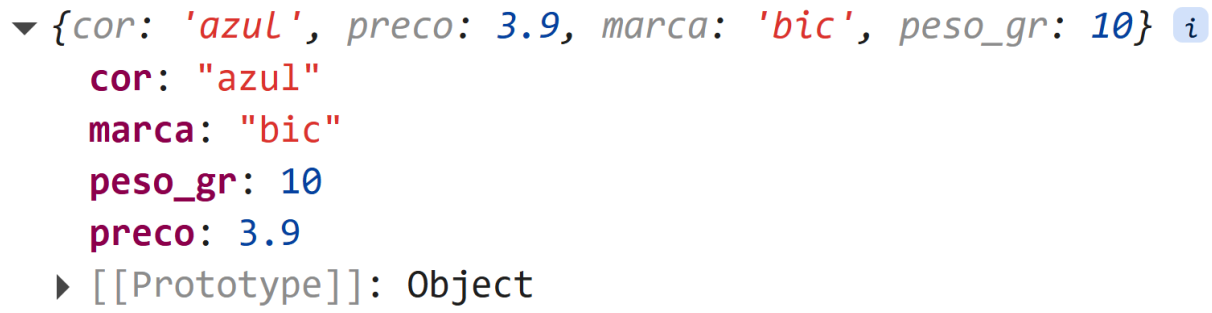


Figura 5.1: Exibição de objeto no console

A exibição de um objeto no console é bastante útil: todas as suas propriedades (ou características) são apresentadas. Um objeto pode ter inúmeras propriedades — tudo depende de como ele foi definido. Por isso, é comum precisarmos acessar apenas uma ou outra característica específica.

Por exemplo, na tela de um produto de um *e-commerce*, são exibidas principalmente as características visuais. Já na tela do carrinho, o foco recai sobre informações como preço e desconto. Para pegar apenas uma característica, basta usar a notação de ponto (.) ou de colchetes (`[]`), como mostrado a seguir:

```
var caneta = {
    cor: 'azul',
    preco: 3.9,
    marca: 'bic',
    peso_gr: 10
}

console.log(caneta.cor);
console.log(caneta["preco"]);
console.log(caneta["marca"]);
console.log(caneta.peso_gr);
```

Também é possível alterar valores das propriedades de um objeto:

```
var caneta = {
    cor: 'azul',
    preco: 3.9,
```

```
    marca: 'bic',
    peso_gr: 10
}

caneta.cor = "vermelho";
caneta["preco"] = 4.10;

console.log(caneta);
```

5.2 Constructor, new e this

Quando temos a necessidade de criar vários objetos que têm os mesmos tipos de características, porém seus valores são diferentes, podemos utilizar funções construtoras que facilitam a criação de vários objetos. Essas funções antecipam a criação de objetos para nós e, com o código `new`, iniciamos suas propriedades. Desta forma:

```
function caneta(cor, preco, marca, peso_gr) {
    this.cor = cor;
    this.preco = preco;
    this.marca = marca;
    this.peso_gr = peso_gr;
}

const caneta1 = new caneta("azul", 3.90, "bic", 10);
const caneta2 = new caneta("vermelha", 3.90, "bic", 10);
const caneta3 = new caneta("preta", 3.90, "masterprint", 12);
const caneta4 = new caneta("verde", 3.60, "masterprint", 12);
console.log(caneta1);
console.log(caneta2);
console.log(caneta3);
console.log(caneta4);
```

Esse código define uma função construtora com as possíveis características que um objeto caneta pode ter. Os parâmetros desta função farão a criação conforme as informações que serão

recebidas. É uma pré-função, que faz a criação dos objetos quando o `new` é acionado.

A palavra-chave this

Essa palavra-chave `this` ("este") é usada para se referir ao elemento atual que está escrevendo o código. Como o exemplo anterior na função construtora, o `this` está associado ao novo objeto instanciado.

Ações e funções com objetos

Até o momento, vimos que objetos carregam características e propriedades. Até fizemos uma analogia com objetos da vida real que também possuem características e agora veremos que também podemos associar ações aos objetos. É possível definir **funções** dentro dos objetos. Essas funções ficam limitadas à execução deste objeto.

Vamos continuar no exemplo do objeto `caneta` e associar a ele uma função que faça sentido, como uma ação `escrever`. Como isso não será uma característica e sim uma função, precisamos acrescentar os `()`. Observe esses objetos que agora têm uma ação `escrever`:

```
function caneta(cor, preco, marca, peso_gr) {
    this.cor = cor;
    this.preco = preco;
    this.marca = marca;
    this.peso_gr = peso_gr;
    this.escrever = function () {
        console.log("Essa é uma caneta que custa " + this.preco);
    }
}

const caneta1 = new caneta("azul", 3.90, "bic", 10);
const caneta2 = new caneta("vermelha", 3.90, "bic", 10);
```

```
caneta1.escrever();
caneta2.escrever();
```

Assim como podemos visualizar individualmente as características de um objeto, também podemos chamar e executar suas ações, ou seja, os métodos associados a ele.

Propriedades e métodos nativos dos objetos

Agora que sabemos o que são as funções dos objetos, podemos conhecer as funções que os próprios objetos do JavaScript têm. Toda vez que criamos uma palavra ou uma variável do tipo texto, o JS faz automaticamente a instância do objeto `String` e assim já tem vários métodos disponíveis, como:

```
var nome_completo = "João da Silva";
console.log(nome_completo.length); //Propriedade que informa quantidade total de caracteres.
console.log(nome_completo.toLowerCase()); //Ação que converte tudo em minúsculo.
console.log(nome_completo.toUpperCase()); //Ação que converte tudo em maiúsculo.
console.log(nome_completo.replace("da Silva", "Pereira")); //Ação que substitui o 1º parâmetro pelo 2º
```

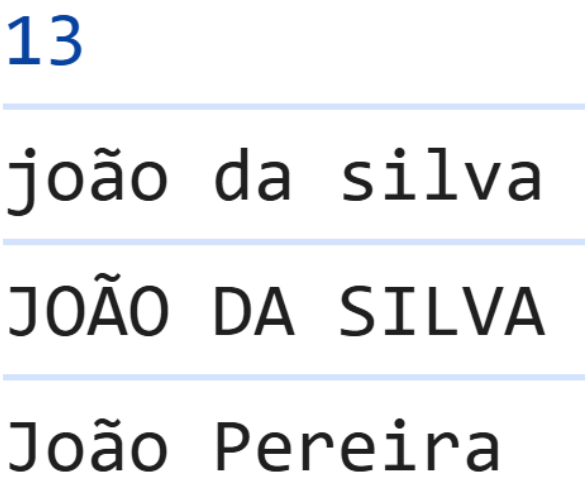


Figura 5.2: Algumas ações nativas do objeto string

5.3 Classes

Na versão ES6 do JavaScript, foi introduzido o conceito de **classes**, também disponível em diversas linguagens de programação. Elas são basicamente modelos para objetos com uma escrita mais simples.

Na criação de uma classe, a função construtora é substituída pelo método `constructor` e as ações nas classes não referenciam propriedades. Elas são declaradas dentro do bloco, sem a palavra `function`. Seguindo boas práticas de desenvolvimento, utilize letra maiúscula para definir uma classe. Confira:

```
class Caneta {
    constructor(cor, preco, marca, peso_gr){
        this.cor = cor;
        this.preco = preco;
        this.marca = marca;
        this.peso_gr = peso_gr;
        this.vendida = false;
    }
    escrever() {
        console.log("Essa é uma caneta que escreve na cor " + this.cor);
    }
    precificar() {
        var lucro = 1;
        var total = this.preco + lucro;
        console.log("O preço desta caneta é " + total);
    }
    vender(){ //Quando essa ação é executada, altera uma característica da
caneta.
        this.vendida = true;
    }
}

const minhaCaneta = new Caneta("azul", 3.9, "Bic", 10);
minhaCaneta.escrever(); //Executando uma ação da classe
minhaCaneta.precificar(); //Executando outra ação da classe
```

```
minhaCaneta.vender(); //Executando outra ação da classe
console.log(minhaCaneta); //Exibe o objeto inteiro no console
```

Em resumo, as classes no JavaScript são uma forma de estruturar objetos personalizados de maneira mais organizada.

5.4 Arrays

É um tipo de dado que serve para armazenar valores de forma sequencial e em lista. Podemos guardar dentro dos arrays uma sequência de números, textos, objetos ou até mesmo uma lista de listas. Os valores dentro dos arrays podem ser diferentes uns dos outros e podemos aumentar ou diminuir essa lista a qualquer momento. Cada item dentro da lista pode ser chamado de elemento, e cada elemento tem um índice. E aqui vai uma das informações mais importantes dos arrays: todo índice (ou `index`) começa no elemento zero. Vejamos uma primeira lista:

Itens do Array:	**Nerak**	**Toni**	**Sushi**	**Gil**	**Beta**
Índice (Index)	0	1	2	3	4

Figura 5.3: Exemplo visual de um array

Neste exemplo, temos um array com os nomes de cinco pets e os índices vão de zero a quatro porque todo array começa no **índice zero**. Se alguém perguntar qual é o primeiro item deste array, a resposta correta é Nerak. E se perguntarem em qual índice está o nome Nerak, a resposta correta é o índice 0. Não se assuste: a maioria das linguagens trabalha com arrays dessa forma. No começo, pode parecer estranho, mas com a prática isso se torna natural.

Para declararmos uma variável deste tipo e atribuir valores, usamos colchetes `[]`. Colocamos os itens dentro de `[]` e separamos os itens desta lista com vírgulas. Vamos fazer alguns:

```
//Array do tipo texto:
var pets = ["Nerak","Toni","Sushi","Gil", "Beta"];

//Array do tipo valores:
var valores = [1.9, 2.99, 3, 19.90]; //Cuidado com arrays de valores: são separados por "." e os elementos separados por ","

//Array do tipo numérico:
var idades = [25, 40, 27, 60, 32];

//Array de tipos variados
var lista_diversa = [35, "Lucas", 1.87, true];

console.log(pets);
console.log(valores);
console.log(idades);
console.log(lista_diversa);
```

Esse código vai produzir a seguinte imagem:

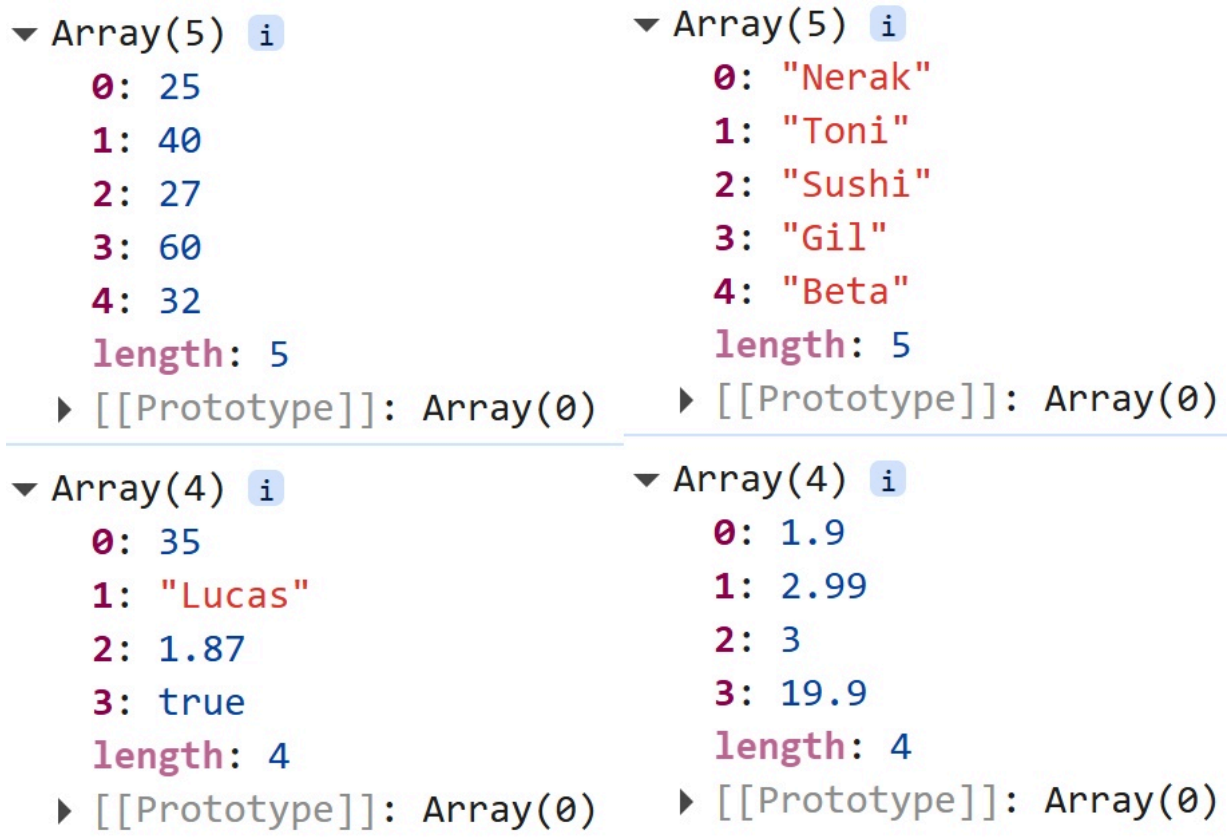


Figura 5.4: Exemplo visual de arrays no console

Veja que interessante como o JavaScript interpretou os arrays criados. No console, ao exibir a lista pets, ele mostra os cinco itens corretamente. Quando clicamos na setinha para expandir a visualização, conseguimos ver todos os elementos detalhadamente, com seus respectivos índices ao lado. O JavaScript associa o índice 0 ao primeiro item da lista, o índice 1 ao segundo item, e assim por diante.

No final da visualização, também aparece uma propriedade chamada `length`, que indica o tamanho do array — nesse caso, 5. Isso confirma que um array com cinco elementos possui índices de 0 a 4. O mesmo comportamento se aplica a qualquer outro array.

Geralmente, utilizamos um item por vez. Seja para buscar ou alterar itens da lista, os índices são fundamentais para essa navegação.

Vamos usar novamente aquele nosso primeiro array para ver, na prática, como acessar e modificar seus itens:

```
//Array do tipo texto:
var pets = ["Nerak","Toni","Sushi","Gil", "Beta"];

console.log(pets[0]); //Busca e exibe o primeiro item do array
console.log(pets[2]); //Busca e exibe o terceiro item do array

pets[1] = "Tony"; //Altera o 2º item do array.
console.log(pets[1]); //Busca e exibe o segundo item do array depois da
alteração
```

Agora que entendemos o conceito de array, vamos falar um pouco sobre seu uso. Vamos supor que você desenvolverá uma tela que vai exibir os clientes cadastrados de uma loja. Os nomes dos clientes estão gravados em um banco de dados e você tem que exibir esses clientes na tela. A informação que vem de qualquer banco de dados é uma lista de itens, o que torna o array perfeito para exibir essa informação na tela.

Assim, um procedimento normal é que informações provenientes de banco de dados sejam colocadas dentro de um array e esse array seja exibido na tela do usuário. Nesse cenário, vamos percorrer todos os itens do array para acessar cada um de seus elementos.

Como isso é uma situação repetitiva, podemos combinar com o uso do **loop** que vimos anteriormente. Definimos um início para o ciclo que sempre será zero (já que todo array começa com índice zero) e definimos o seu final com a propriedade `length` do array, pois, como vimos, essa propriedade informa a quantidade total de itens. Olha que legal:

```
var valores = [10, 15, 20, 22];
for (var i = 0; i<valores.length; i++){
    console.log(valores[i]);
}
```

Nesse código, indicamos que será feito um loop que começa em zero e vai até o tamanho total do array valores (`valores.length`) e a atualização será de um em um (`i++`). Ou seja, estamos usando a variável `i` de índice. Esse loop exibirá todos os itens do array, um a cada iteração. Quando atingir o último item, o loop será encerrado.

5.5 Métodos e propriedades dos Arrays

O JavaScript oferece diversos métodos e propriedades que facilitam o trabalho com arrays. Vamos conferir os mais utilizados.

Length

O primeiro deles é o que já conhecemos: o `length` , que nos permite identificar quantos elementos há no array. É muito comum usarmos essa propriedade para definir o limite nos `loops` que envolvem arrays.

push() e unshift()

O método `push` nos permite acrescentar elemento a um array já existente. Quando usado, ele adiciona o novo elemento ao **final do array**. Se quisermos adicionar ao início, então temos que usar o `unshift` .

```
var disciplinas = ["Matemática", "Português", "Inglês"];

//Adicionado no final, ficando ["Matemática", "Português", "Inglês",
"Química"]
disciplinas.push("Química");

//Adicionado no começo, ficando ["Física", "Matemática", "Português",
```

```
"Inglês", "Química"]
disciplinas.unshift("Física");
```

shift() e pop()

De forma inversa, existem os métodos que removem itens do array. Para eliminar o primeiro, usamos `shift()`, e para eliminar o último, usamos `pop()`.

```
var disciplinas = ["Matemática", "Português", "Inglês"];

//Remove o primeiro item ficando ["Português", "Inglês"]
disciplinas.shift();

//Remove o último item ficando ["Português"]
disciplinas.pop();
```

splice()

Este método permite eliminar um ou mais elementos de um array e em qualquer posição. Ele funciona com dois parâmetros: o primeiro indica o índice inicial da remoção, e o segundo, a quantidade de elementos a serem excluídos a partir desse índice.

```
var disciplinas = ["Matemática", "Português", "Inglês", "Quimica",
"Física"];
disciplinas.splice(1, 3); //Inicia no segundo item e remove três
elementos.
console.log(disciplinas); //Resultado será ["Matemática", "Física"]
```

join()

Este método faz com que todos os itens do array virem um único texto separado pelo valor que indicamos no único parâmetro:

```
var disciplinas = ["Matemática", "Português", "Inglês"];
console.log(disciplinas.join(" - ")); //Resultado será Matemática -
Português - Inglês
```

concat()

Este método nos permite combinar dois arrays em um único:

```
var disciplinas = ["Matemática", "Português", "Inglês"];
var provas = ["Derivada", "Integral", "Gramática", "Verbo to be"];
var todos = disciplinas.concat(provas);
console.log(todos); //Resultado será um array com 7 itens: ['Matemática',
'Português', 'Inglês', 'Derivada', 'Integral', 'Gramática', 'Verbo to be']
```

slice()

Retorna uma parte do array para um novo array. Ele também funciona com dois parâmetros, sendo os índices de começo e fim das posições. Neste método o array original não é alterado.

```
var disciplinas = ["Matemática", "Português", "Inglês", "Quimica",
"Física"];
var humanas = disciplinas.slice(1, 3);
console.log(humanas); //Resultado será: ['Português', 'Inglês']
```

indexOf()

Este método nos permite saber o índice de um elemento no array. Se o elemento que procuramos não existir, essa função nos retorna `-1`.

```
var disciplinas = ["Matemática", "Português", "Inglês", "Química",
"Física"];
console.log(disciplinas.indexOf("Português")); //Será exibido 1
console.log(disciplinas.indexOf("Química")); //Será exibido 3
console.log(disciplinas.indexOf("História")); //Será exibido -1
```

Se houver elementos repetidos no array, o `indexOf()` sempre retorna o índice da primeira ocorrência.

```
var disciplinas = ["Inglês", "Matemática", "Inglês", "Física"];
console.log(disciplinas.indexOf("Inglês")); // Será exibido 0, que é a
```

```
primeira ocorrência
```

Se você quiser encontrar outras ocorrências do mesmo elemento, pode combinar `indexOf()` com laços de repetição (`for` , `forEach`) ou utilizar métodos mais avançados, como `filter()` . Veremos esses elementos mais adiante.

reverse()

Este método inverte a ordem dos elementos de uma lista:

```
var disciplinas = ["Matemática", "Português", "Inglês"];
disciplinas.reverse();
console.log(disciplinas); //Resultado: ["Inglês", "Português",
"Matemática"]
```

sort()

Este método é útil para ordenar os elementos do seu array. Arrays com valores serão ordenados de forma crescente e arrays com textos serão ordenados alfabeticamente.

```
var disciplinas = ["Matemática", "Português", "Inglês"];
disciplinas.sort();
console.log(disciplinas); //Resultado: ["Inglês", "Matemática",
"Português"]
```

5.6 Arrays com objetos

Essa combinação gera estrutura de dados complexas e com as ferramentas que vimos até agora podemos manipular essas estruturas complexas de forma precisa e ordenada. Vejamos:

```
var carrinho = [
    { id: 1, produto: "Camiseta", cor: "Azul", tamanho: "M", preco: 59.90
```

```
},
    { id: 2, produto: "Calça", cor: "Verde", tamanho: "G", preco: 109.90
},
    { id: 3, produto: "Tênis", cor: "Preto", tamanho: "42", preco: 229.90
},
    { id: 4, produto: "Camiseta", cor: "Branca", tamanho: "G", preco:
59.90 },
    { id: 5, produto: "Calça", cor: "Azul", tamanho: "M", preco: 109.90 }
];
```

Temos um array de objetos (uma lista de produtos e suas características). Essa lista tem cinco itens e é um array muito comum de ser encontrado em uso em qualquer loja virtual depois de o usuário encher seu carrinho de compras.

Vamos imaginar uma página de favoritos, em que não é necessário exibir o código e nem a cor do item. Então poderíamos exibir todos os itens favoritos desta forma:

```
for(var i = 0; i < carrinho.length; i++){
    console.log("Item favoritado: " + carrinho[i].produto + " na cor " +
carrinho[i].cor);
}
```

Já quando o usuário for encerrar a compra, é necessário exibir o preço e podemos usar os mesmos itens, porém com as propriedades diferentes:

```
for(var i = 0; i < carrinho.length; i++){
    console.log("Item: " + carrinho[i].produto + " no tamanho " +
carrinho[i].tamanho + " preço: " + carrinho[i].preco);
}
```

Essa estrutura é perfeita para diversas situações e é o formato mais utilizado para transferir e integrar informações nos sistemas. Afinal, em uma única estrutura conseguimos trabalhar uma lista de dados com diversas características.

5.7 Algumas maravilhas do JavaScript

As funções de Ordem Superior podem parecer um pouco abstratas à primeira vista, mas entender esse conceito nos ajuda a reconhecer e aproveitar melhor diversas funcionalidades já disponíveis do JavaScript — como `sort()`, `map()`, `filter()` e muitas outras.

Essas funções nativas são exemplos poderosos de como a linguagem implementa o conceito de abstração de forma simples para a pessoa desenvolvedora. Elas encapsulam lógicas complexas em estruturas fáceis de usar, o que nos permite escrever códigos mais limpos e eficientes.

A seguir, vamos explorar mais algumas funcionalidades que o JavaScript oferece prontas para uso — verdadeiras maravilhas que nos poupam tempo e esforço.

Math

O JavaScript possui, de forma nativa, um objeto chamado `Math`, que reúne diversas funcionalidades prontas para realizar operações matemáticas. Afinal, ao programar, é muito comum lidarmos com cálculos como um controle de vendas ou um caixa de loja.

Para facilitar essas operações, o objeto `Math` oferece um compilado de ferramentas para operações matemáticas. Vamos iniciar por algumas propriedades:

```
var pi = Math.PI;
var euler = Math.E;

console.log(pi); //3.141592653
console.log(euler); //2.7182818
```

Também temos algumas funções interessantes:

Math.min() e Math.max()

Esses métodos recebem valores numéricos e, como o nome sugere, retornam o valor mínimo e máximo entre eles:

```
console.log( Math.max(-10, 0, -7, 10, 40, 20) ); // Resultará 40
console.log( Math.min(-10, 0, -7, 10, 40, 20) ); // Resultará -10
```

Math.ceil(), math.floor() e math.round()

Esses métodos recebem um valor numérico com casas decimais e fazem o arredondamento de formas diferentes, de acordo com a regra de cada um:

```
var pi = Math.PI;
console.log(pi); //Valor padrão de PI: 3.141592653
console.log(Math.ceil(pi)); //Resultará 4 pois retorna o número inteiro superior mais próximo
console.log(Math.floor(pi)); //Resultará 3 pois retorna o número inteiro inferior mais próximo
console.log(Math.round(pi)); //Resultará 3 pois retorna o valor de um número arredondado, o número inteiro mais próximo
```

Math.sqrt()

Essa funcionalidade retorna a raiz quadrada de um número. Se o número for negativo, o método retorna *NaN (Not a Number)*.

```
console.log(Math.sqrt(81)) // 9
console.log(Math.sqrt(2)) // 1.414213562373095
console.log(Math.sqrt(-10)) // NaN
```

Math.random()

Método que gera um número aleatório entre 0 e 1. Assim, ao usar a multiplicação, você consegue gerar uma faixa qualquer de números aleatórios.

```
console.log(Math.random() * 10) // Número entre 0 e 10
console.log(Math.random() * 100) // Número entre 0 e 100
console.log(Math.random() * 1000) // Número entre 0 e 1000
```

Date

Outra classe que vai facilitar nossa vida na programação JS é a `Date`, pois em algum momento, ao lidar com dados, precisaremos manipular datas. Seja data de nascimento, data de cadastro, data da compra etc. Essa classe foi projetada para representar datas e possui um vasto repertório. O primeiro uso que veremos é como obter a data atual no seu código:

```
console.log( new Date() ); //Resultará Tue Feb 25 2025 17:45:38 GMT-0300 (Horário Padrão de Brasília)
```

Como não passamos nenhum parâmetro para essa função, ele retornou a data atual completa em inglês — inclusive com o dia da semana sendo *Mon, Tue, Wed, Thu, Fri, Sat e Sun* (traduzindo: Seg, Ter, Qua, Qui, Sex, Sat, Dom). Observe que ele também puxou a data do fuso horário em que estou, o "GMT-0300 Horário Padrão de Brasília". Esse objeto `Date` possui um construtor que nos permite criar objetos com datas diferentes da atual. Ele pode receber parâmetros nesta ordem: ano, mês, dia, hora, minuto, segundos e milissegundos. Todos usando o tipo numérico.

```
var niver = new Date(2025, 1, 6);
var virada = new Date(2025, 11, 31, 23, 59, 59);

console.log(niver); //Resultará: Thu Feb 06 2025 00:00:00 GMT-0300 (Horário Padrão de Brasília)
console.log(virada); //Resultará: Wed Dec 31 2025 23:59:59 GMT-0300 (Horário Padrão de Brasília)
```

No exemplo, definimos uma data com os parâmetros que representam o dia e o resultado zerou os horários. Já em um outro exemplo com mais parâmetros, a data vem com os horários também. Uma observação importante são os meses. Observe que o mês de Fevereiro foi descrito pelo segundo parâmetro com valor 1.

Isso acontece porque os meses no JS são definidos de 0 a 11, sendo 0 = Janeiro e 11 = Dezembro.

Também é possível criar uma data a partir de uma string com formato específico:

```
var virada = new Date("December 31, 2025 23:59:59");
console.log(virada); //Resultará: Thu Feb 06 2025 00:00:00 GMT-0300 (Horário Padrão de Brasília)
```

Esse formato exige um pouco mais de cuidado, considerando que você tem que lembrar exatamente o formato da string, o nome do mês em inglês, exatamente onde fica a vírgula... Na minha opinião, o modo anterior é mais fácil, mas você pode ter mais afinidade com esse formato.

O **Date** também tem alguns outros métodos prontos para exibição de datas ou captura de parte de uma data. Como os exemplos a seguir:

```
const niver = new Date(2025, 1, 6);
console.log(niver.toDateString()) // Thu Feb 06 2025
console.log(niver.toLocaleString()) // 06/02/2025, 00:00:00
console.log(niver.toLocaleDateString()) // 06/02/2025
console.log(niver.toTimeString()) // 00:00:00 GMT-0300 (Horário Padrão de Brasília)
console.log(niver.toLocaleTimeString()) // 00:00:00
console.log(niver.toString()) // Thu Feb 06 2025 00:00:00 GMT-0300 (Horário Padrão de Brasília)
console.log(niver.toUTCString()) // Thu, 06 Feb 2025 03:00:00 GMT
```

Outro uso interessante do objeto `Date` é o cálculo do intervalo entre duas datas. Em diversas situações, pode surgir a necessidade de saber quantos dias (ou horas, minutos) se passaram entre dois momentos. O JavaScript facilita isso: podemos subtrair uma data da outra, e o resultado será a diferença entre elas em milissegundos.

```
const virada = new Date(2025, 11, 31, 23, 59, 59); // Data da virada do ano (31 de dezembro de 2025, 23h59m59s)
```

```
const data = new Date(); // Data e hora atual

const diffMs = virada - data; // Diferença em milissegundos
const diffSegundos = diffMs / 1000; // Converte milissegundos para
segundos
const diffMinutos = diffSegundos / 60; // Converte segundos para minutos
const diffHoras = diffMinutos / 60; // Converte minutos para horas
const diffDias = diffHoras / 24; // Converte horas para dias

console.log("Faltam aproximadamente: " + Math.floor(diffDias) + " dias ou
" +
    Math.floor(diffHoras) + " horas ou " +
    Math.floor(diffMinutos) + " minutos ou " +
    Math.floor(diffSegundos) + " segundos para a virada do ano.");
```

Aprendemos a manipular objetos, arrays, textos e datas. Esses conhecimentos são indispensáveis para trabalhar com dados reais em aplicações web. Mas o JavaScript ainda tem mais truques! Chegou o momento de conhecer algumas das técnicas mais poderosas que a linguagem oferece para deixar nosso código mais inteligente, conciso e moderno. Vejo você no próximo capítulo!

Capítulo 6
Abstrações e lógicas avançadas

Neste capítulo, vamos explorar funções de Ordem Superior, operadores avançados e técnicas modernas que tornam o JavaScript uma linguagem altamente expressiva. Não se preocupe se alguns conceitos pareçam complexos no início — com os exemplos práticos, tudo ficará mais claro.

6.1 Funções de Ordem Superior

As funções de Ordem Superior ou, em inglês, *Higher Order Functions*, são maravilhas do JavaScript que facilitam nossa vida quando estamos trabalhando com arrays e objetos. Essas funções têm um conceito bastante complexo, mas a sua aplicação prática é fácil. Então, preciso explicar o que é, mas não se preocupe se o entendimento ficar abstrato na sua mente. Na hora em que você aplicar, tudo ficará claro como água. Vamos lá!

Abstração

No contexto de funções de Ordem Superior, a abstração é como resumir um grupo complexo de instruções em uma pequena função que especifica a ação que ela pode executar. Essas abstrações ocultam detalhes sobre algumas operações executadas, mas resolvem grandes problemas para nós em um nível mais alto (ou em um nível mais abstraído). Como programadores(as), estamos sempre criando abstrações para poder pensar em um nível mais alto. Tentamos construir soluções complexas para os desafios que surgem. No JavaScript, existem diversos métodos nativos (já embutidos nele) que oferecem soluções para problemas recorrentes.

Essa explicação foi basicamente para mostrar que é possível criar funções que retornam outras funções! Sei que isso pode parecer confuso no começo. Se funções são como "solucionadoras de problemas", por que retornar outra função em vez de já devolver a solução?

A resposta está justamente na ideia de abstração: às vezes, queremos criar um comportamento que será definido mais tarde, ou que poderá variar conforme o contexto. Nesse caso, uma função que retorna outra função nos ajuda a montar soluções mais flexíveis e reutilizáveis. É como criar uma "fábrica de soluções", capaz de gerar funções sob medida para diferentes situações. Vamos ver um primeiro exemplo.

```
function Escolher(operacao){
    if(operacao == "subtrair"){
        return (a, b) => a - b;
    }
    else if(operacao == "somar"){
        return (a, b) => a + b;
    }
    else if(operacao == "multiplicar"){
        return (a, b) => a * b;
    }
}

let multiplicar = Escolher("multiplicar");
let somar = Escolher("somar");

console.log( multiplicar(10, 7) ); //Resultará 70
console.log( somar(10, 5) ); //Resultará 15
```

Olha que coisa doida: criamos uma função chamada `Escolher` que, dependendo da condição (`if`) atendida, retorna outra função — inclusive, pode ser até uma **arrow function**!

Essa função retornada pode ser atribuída a uma variável, que, ao ser utilizada, é como se estivéssemos chamando a função diretamente:

basta usar os parênteses e passar os parâmetros necessários. E a coisa ainda fica mais interessante: se uma função pode retornar outra, também é possível que uma função seja passada como parâmetro para outra. Veja o exemplo:

```
function UmPorUm(array, funcao) {
    for (var i=0; i< array.length; i++) {
        funcao(array[i]);
    }
}

const pets = ["Nerak", "Gil", "Toni", "Sushi"];
UmPorUm(pets, console.log);
```

Enviamos a função `console.log` porque já a conhecemos bem. Olha que interessante: passamos uma função por parâmetro para ser executada dentro da outra função com o nome do parâmetro que demos. UAU! Isso sim é abstração!

Tudo bem. Prometo que vou parar com as loucuras neste tópico.

Toda essa explicação foi apenas para que você entendesse um mecanismo complexo existente no JavaScript, que é pouco utilizado. O que utilizamos com frequência são as funções prontas que o JS já oferece e que recorrem a esse mecanismo.

A partir de agora, vou mostrar algumas funções e exemplos **prontinhos** que usam esse mecanismo complexo — e que vão facilitar muito a nossa vida.

6.2 Métodos nativos de busca e transformação

O JS tem várias funções de Ordem Superior nativamente incorporadas. Veremos algumas dessas funções que trabalham com arrays. Os métodos mais úteis e legais atuam em todos os elementos de um array e recebem uma função por parâmetro. Cada

uma das funções a seguir foi pensada em solucionar problemas comuns com arrays.

forEach()

O `forEach` é parecido com o exemplo `UmPorUm`. Na verdade, fiz esse método propositalmente parecido para que você tenha uma noção de que não precisa saber o que está dentro daquela função de Ordem Superior. Basta saber usar essas funções.

```
const pets = ["Nerak", "Gil", "Toni", "Sushi"];

pets.forEach((nome) => {
    console.log(nome)
});
```

Este exemplo faz um loop sobre o array de referência, executando a função que enviamos por parâmetro para cada elemento do array `pets`. É exatamente o que a função `UmPorUm`, do exemplo anterior, faz — porém sem a necessidade de sabermos como a função `forEach` foi implementada. Nós simplesmente a utilizamos.

find()

Agora vamos conhecer o `find` que recebe uma função de comparação por parâmetro. A função captura o elemento que está percorrendo e retorna verdadeiro ou falso conforme a comparação executada.

```
const pets = [
    {nome: 'Nerak', idade: 10, raca: "Shitzu"},
    {nome: 'Gil', idade: 4, raca: "Periquito Australiano"},
    {nome: 'Toni', idade: 3, raca: "Shitzu"},
    {nome: 'Sushi', idade: 1, raca: "Basset"}
];

const resultado1 = pets.find((o) => o.raca == "Basset");
const resultado2 = pets.find((o) => o.idade > 5);
const resultado3 = pets.find((o) => o.nome == "Gil");
```

```
console.log(resultado1); //Resultará {nome: 'Sushi', idade: 1, raca:
"Basset"}
console.log(resultado2); //Resultará {nome: 'Nerak', idade: 10, raca:
"Shitzu"}
console.log(resultado3); //Resultará {nome: 'Gil', idade: 4, raca:
"Periquito Australiano"}
```

É importante observar que essa função retornará apenas o primeiro elemento encontrado na condição. Caso haja mais elementos que atendem a condição de comparação, apenas o primeiro encontrado será exibido.

filter()

O `filter` é muito parecido com o `find`. Ele também recebe uma função que compara por parâmetro. A diferença é que o `filter` retorna todos os elementos que atendam a essa função.

```
const pets = [
    {nome: 'Nerak', idade: 10, raca: "Shitzu"},
    {nome: 'Gil', idade: 4, raca: "Periquito Australiano"},
    {nome: 'Toni', idade: 3, raca: "Shitzu"},
    {nome: 'Sushi', idade: 1, raca: "Basset"}
];

const resultado = pets.filter((o) => o.raca == "Shitzu");

console.log(resultado);
//Resultará {nome: 'Nerak', idade: 10, raca: "Shitzu"}, {nome: 'Toni',
idade: 3, raca: "Shitzu"},
```

map()

O `map` cria um novo array com os elementos do array original transformando os itens de acordo com a operação enviada por parâmetro.

```
const pets = [
    {nome: 'Nerak', idade: 10, raca: "Shitzu"},
    {nome: 'Gil', idade: 4, raca: "Periquito Australiano"},
    {nome: 'Toni', idade: 3, raca: "Shitzu"},
    {nome: 'Sushi', idade: 1, raca: "Basset"}
];

const resultado = pets.map((item) => {
    if(item.raca == 'Periquito Australiano') {
        return {categoria: "Passaro", nome: item.nome };
    }
    else{
        return {categoria: "Cachorro", nome: item.nome };
    }
});

console.log(resultado);
```

O resultado exibirá:

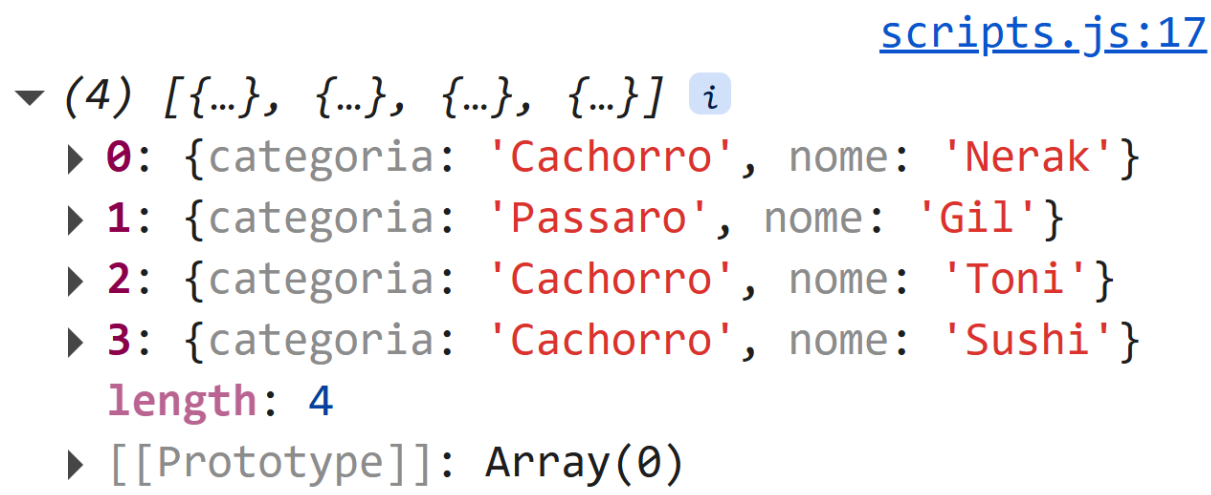


Figura 6.1: Exemplo novo array criado com map

Neste tópico, eu trouxe um exemplo maior para visualizarmos que podemos passar por parâmetros várias operações e ainda construir um novo array totalmente diferente do original. Esse novo array usa a característica da raça para fazer uma comparação e criar um novo array com uma nova característica de categoria. Também é possível

aproveitar outras características, como nesse exemplo em que aproveitamos o nome.

sort()

Considero como um dos mais úteis. Ele permite ordenar um array a partir de um critério que definimos. Esse método recebe uma função de comparação por parâmetro que recebe dois elementos do array. Essa função retorna valores numéricos sendo 1, -1 ou 0, que indica qual elemento se posiciona antes ou depois.

```
const pets = [
    {nome: 'Nerak', idade: 10, raca: "Shitzu"},
    {nome: 'Gil', idade: 4, raca: "Periquito Australiano"},
    {nome: 'Toni', idade: 3, raca: "Shitzu"},
    {nome: 'Sushi', idade: 1, raca: "Basset"}
];

pets.sort((a,b) => {
    if(a.nome > b.nome){
        return 1;
    } else {
        return -1;
    }
});

console.log(pets);
/* Resultará:
{nome: 'Gil', idade: 4, raca: 'Periquito Australiano'},
{nome: 'Nerak', idade: 10, raca: 'Shitzu'},
{nome: 'Sushi', idade: 1, raca: 'Basset'},
{nome: 'Toni', idade: 3, raca: 'Shitzu'}
*/
```

O exemplo mostra como ordenar usando uma característica de texto. Se quiser inverter a ordenação de forma que o array seja ordenado também pelo nome, mas de forma decrescente, é só alterar o sinal da comparação de `>` para `<` na estrutura `if`.

Se sua comparação for feita por uma característica numérica, a ordenação é mais simples ainda. Veja:

```
const pets = [
    {nome: 'Nerak', idade: 10, raca: "Shitzu"},
    {nome: 'Gil', idade: 4, raca: "Periquito Australiano"},
    {nome: 'Toni', idade: 3, raca: "Shitzu"},
    {nome: 'Sushi', idade: 1, raca: "Basset"}
];

pets.sort((a, b) => a.idade - b.idade);
console.log(pets); //Exibirá o array ordenado pela característica idade de forma crescente.

pets.sort((a, b) => b.idade - a.idade); //Fazer comparação dos parâmetros de forma inversa para invertermos a comparação.
console.log(pets); //Exibirá o array ordenado pela característica idade de forma decrescente.
```

A aplicação do método `sort` é irreversível. A partir do momento em que seu array for ordenado, não é possível voltar à ordenação original.

6.3 Operadores avançados

Passamos por diversos métodos e, com o conhecimento técnico que temos, podemos falar dos operadores (ou **Sugar Syntax**). Esses operadores servem para simplificar tarefas complexas.

Operador ++

Esse operador serve para aumentar em 1 o valor de qualquer variável numérica. Ele só funciona em variáveis do tipo inteiro. Já vimos esse operador em funcionamento nos laços de repetição quando trabalhamos com o `for` . Vamos conferir alguns exemplos com esse operador:

```
var numero = 2;
numero = numero + 1; //Forma normal de aumentar mais 1, deixando a variável numero com resultado 3.
numero++; //Operador ++ aumenta mais 1 e a variável numero fica com 4.
numero += 1; //Outra forma simplificada de aumentar 1. Agora a variável numero é 5.
```

Operador ternário

Este operador é uma simplificação da estrutura `if/else`. Quando temos comparações simples para fazer, esse operador ternário é bem-vindo. A seguir, vou fazer a mesma comparação duas vezes. Uma usando o `if/else` comum e a outra usando o operador ternário:

```
let idade = 20;
if(idade > 18){
    alert("Pode tirar CNH");
}
else {
    alert("Ainda não pode ter CNH");
}

//O trecho a seguir é um resumo do if/else acima
(idade > 18) ? alert("Pode tirar CNH") : alert("Ainda não pode ter CNH")
```

A condição no operador ternário é idêntica à da estrutura `if/else`. Na sequência, o "?" indica o início desta estrutura. Colocamos o comando em caso de condição verdadeira; depois, ":" recebe os comandos em caso de condição falsa.

Resumindo:

```
(condicao) ? comandos_se_verdadeiro : comandos_se_falso;
```

A ideia do operador ternário é realmente para condições muito curtas, com pouquíssimos comandos, pois, em comparações que necessitem de mais códigos, a estrutura `if/else` ainda é mais útil, considerando a sua organização dentro dos delimitadores `{}`.

Operador lógico AND

Você já viu o operador `&&` sendo utilizado anteriormente neste livro, especialmente dentro de instruções `if`. Lá, ele servia como um atalho para executar algo somente se uma condição fosse verdadeira. Agora, vamos explorar um comportamento mais completo e característico do JavaScript: o uso do operador `&&` para **retornar valores**. Nesse caso, ele pode ser usado fora de estruturas condicionais, como em atribuições, expressões e retornos diretos.

Esse operador também é uma redução de uma condicional que tenta retornar algo somente se a condição for verdadeira. Reduz um `if` com um único bloco de execução:

```
var pessoa = {
    nome: 'Lucas',
    idade: 35
}

const hora_registro = pessoa.idade >= 18 && new Date();
console.log(hora_registro); //Retorna data atual
```

Segue um resumo da lógica do `&&`: se a primeira condição for `false`, o operador && não verifica o segundo valor e retorna `false` imediatamente; se a primeira condição for `true`, o operador `&&` retorna o **segundo valor**.

Operador lógico OR

Assim como o operador `&&`, o `||` também já apareceu em exemplos anteriores — mas agora veremos como ele pode ser usado para **retornar o primeiro valor verdadeiro** entre dois operandos. Esse uso é muito comum quando queremos definir um valor padrão, caso o primeiro não esteja disponível (por exemplo, esteja `null` ou `undefined`).

Este operador é semelhante ao anterior com diferença de que consiste em dois operandos e não uma única condição. Se for diferente de `falsy` (falsy é diferente de 0, `null`, `undefined`, `NAN`, `false`, ou texto vazio) o operador `OR` retorna o operando 1. Caso contrário retorna o operador 2.

```
alert(0 || "Falsy"); //Falsy
alert(10 || "Falsy"); //10
alert(null || "Falsy"); //Falsy
alert(undefined || "Falsy"); //Falsy
alert("Olá Dev!" || "Falsy"); //Olá Dev!
alert("" || "Falsy");//Falsy
alert(NaN || "Falsy");//Falsy
alert(true || "Falsy"); //true
alert(false || "Falsy");//Falsy
```

Também podemos usar em condições de atribuição:

```
var pessoa1 = {
    nome: 'Lucas',
    idade: 35
}
var pessoa2 = null;
alert(pessoa1 || "Essa pessoa não existe"); //Retornará o objeto pessoa1
alert(pessoa2 || "Essa pessoa não existe"); //Retornará "Essa pessoa não
existe"
```

Operador nullish coalescing

Este operador (`??`) funciona da mesma forma que o operador anterior, a diferença é que aceita mais valores como verdadeiros.

```
alert(0 ?? "Nullish"); //0
alert(10 ?? "Nullish"); //10
alert(null ?? "Nullish"); //Nullish
alert(undefined ?? "Nullish"); //Nullish
alert("Olá Dev!" ?? "Nullish"); //Olá Dev!
alert("" ?? "Nullish");// ""
alert(NaN ?? "Nullish");// NaN
alert(true ?? "Nullish"); //true
alert(false ?? "Nullish");//false
```

Acesso condicional a propriedades de um objeto

Quando tentamos acessar um objeto que não existe, o JS naturalmente gera erro. Mas haverá casos em que não teremos certeza se o objeto existirá e o operador `?` nos ajuda a melhorar o controle sobre esses tipos de erros. Veja um bom exemplo:

```
var pet = null;
console.log(pet.nome); //Isso gera erro, pois, neste momento, ainda não
existe a característica nome neste objeto.

pet = {
    nome: "Nerak",
    idade: 10,
    raca: "Shitzu",
    vacinas_tomadas: {
        vacina1: "V10"
    }
}

//Isso verifica se existe a característica nome e caso não exista, temos
uma mensagem padrão
console.log(pet?.nome || "Característica inexistente."); //Resultará:
Nerak
console.log(pet?.vacinas_tomadas?.vacina1 || "Característica
inexistente."); //Resultará: Nerak
```

```
console.log(pet?.vacinas_tomadas?.vacina2 || "Característica
inexistente."); //Resultará: Característica inexistente
```

6.4 Desestruturação

Essa técnica consiste em permitir declarar variáveis que podem armazenar características de um objeto de forma rápida e direta. Fazemos isso quando vamos usar valores das características, mas sem correr o risco de alterar as propriedades do objeto.

```
var pet = {
    nome: "Nerak",
    idade: 10,
    raca: "Shitzu",
}

const { nome, idade, raca } = pet;
console.log(nome, idade, raca); //Resultará Nerak 10 Shitzu
```

As chaves `{}` à esquerda do operador de atribuição `=` significa que estamos desestruturando um objeto. Também vale observar que `nome, idade, raca` (variáveis) são diferentes de `pet.nome`, `pet.idade` e `pet.raca` (que são propriedades do objeto `pet`). Se tentarmos desestruturar uma característica inexistente, obteremos `undefined`.

Também podemos ter a necessidade de acessar características mais internas dentro de um objeto. Por exemplo, desestruturar uma propriedade que também é um objeto. Podemos fazer seguindo o mesmo padrão.

```
var pet = {
    nome: "Nerak",
    idade: 10,
    raca: "Shitzu",
    vacinas_tomadas: {
        vacina1: "V10"
```

```
    }
}

var {nome, vacinas_tomadas: { vacina1 } } = pet;

console.log(nome); //Nerak
console.log(vacina1); //V10
```

Como `vacinas_tomadas` é um objeto, desestruturamos dele a característica `vacina1`, dentro da desestruturação do objeto `pet`.

6.5 Spreads

Esta ferramenta nos permite espalhar um array ou objeto. Ou seja, mudar a forma como apresentamos esse array ou objeto. Começando pelo `spread` de arrays, quando passamos um array por parâmetro, ele vai inteiro. Se utilizarmos "...", estamos enviando um `spread` desse array e o resultado são todos os itens do array separados:

```
var pets = ["Nerak", "Gil", "Toni", "Sushi", "Beta"];
console.log(pets); //Assim passamos o array inteiro
console.log(...pets); //Assim passamos os itens soltos desta forma: Nerak Gil Toni Sushi Beta
```

Com `spread`, todos os elementos são enviados como parâmetros individuais. Isso é útil quando trabalhamos com funções que não recebem array, mas sim uma série de parâmetros individuais, como é o caso do `Math.max()` e do `Math.min()`.

Já o `spread` de objetos é útil quando queremos replicar ou modificar estruturas de objetos, pois nos permite listar todas as características e valores e, em seguida, modificar as informações que queremos.

```
var pet = {
    nome: "Nerak",
```

```
    idade: 10,
    raca: "Shitzu",
}

var pet2 = {
    ...pet,
    idade: 11,
    peso: 9.8,
    pelagem: "branca"
}

console.log(pet2); // {nome: 'Nerak', idade: 11, raca: 'Shitzu', peso:
9.8, pelagem: 'branca'}
```

No exemplo, mantivemos o objeto `pet` intacto e depois criamos um novo objeto `pet2` a partir do primeiro e modificamos a propriedade `idade` e adicionamos duas novas propriedades.

Nossa, está sendo uma grande jornada! Nesse momento, temos um domínio avançado de JavaScript. Entendo que para a próxima etapa precisamos agregar esse conhecimento aos componentes das páginas criadas com o HTML e CSS. Para isso, precisamos **linkar** as tecnologias. Bora lá?!

Capítulo 7
JavaScript na Internet

No capítulo anterior, fomos muito a fundo no JavaScript. Conhecemos diversos conceitos dos mais básicos aos mais avançados. Mas você deve ter percebido (e talvez também tenha ficado um pouco cansado) que todos os exemplos foram feitos ou com `console.log` ou com `alert`.

Em outras palavras, mal usamos o navegador. Não colocamos exemplos em tela. Aliás, não criamos telas. Foi só a *teoria-técnica-execução*, tudo visível em um console que só pessoas desenvolvedoras acessam. Pois bem, agora que temos todo o poder do JavaScript em mente, podemos começar a entender como ele se aplica às propriedades do navegador.

Então, você está me dizendo que tudo o que vimos do JavaScript é possível executar também em outras plataformas além do navegador? Sim. Um exemplo disso é o Node.js, mas isso é papo para outro livro. Como comentei lá no começo do livro, íamos aprender toda a lógica de programação, aplicável inclusive em outras linguagens.

O que estou dizendo é que até aqui você aprendeu o domínio lógico de programação. A partir de agora, o livro começa a ficar mais delimitado. Vamos abordar técnicas precisas do JavaScript aplicadas aos navegadores. E o foco será ainda maior no próximo capítulo, quando enfim entraremos nos Dashboards.

Pois bem, vamos às aplicações específicas do JavaScript na Web.

7.1 Document Object Model (DOM)

O DOM é a interface que representa como os documentos HTML (ou até mesmo XML) são lidos pelo seu navegador. Nós o vimos brevemente no início do livro, mas agora abrangeremos o assunto com mais profundidade e de forma mais técnica.

Após o navegador ler seu documento HTML, ele cria um objeto que é a representação estruturada do seu documento e define os meios pelos quais tudo será acessado. Esse documento (que podemos resumir como seu site HTML) pode ser consultado e manipulado com JavaScript.

Para cada página carregada no navegador, existe uma interface (o DOM). Essa interface serve como ponto de entrada para o conteúdo da página HTML, composta pelas tags. Cada tag HTML é um objeto que podemos chamar de nó. E as tags aninhadas são chamadas "nós filhos" da tag do "nó pai", como a seguir:

```
<body> <!-- Nó pai -->
    <h1>Título</h1> <!-- h1 é nó filho da tag body -->
    <p>Um exemplo de parágrafo com um  <!-- p é nó filho da tag body -->
        <a href="https://espaco.dev/">link aqui</a>.  <!-- a é nó filho da
tag p -->
    </p>
</body>
```

Para vincularmos um arquivo JavaScript, precisamos fazer referência a ele no final do `body` , lembra? Algo assim:

```
<body>
    <h1>Título</h1>
    <p>Um exemplo de parágrafo com um
        <a href="https://espaco.dev/">link aqui</a>.
    </p>
    <script src="js/scripts.js"></script> <!-- Estamos chamando um arquivo
de script -->
</body>
```

A partir do momento em que efetivamos essa chamada de um (ou mais) arquivo(s) script(s), temos acesso, via JavaScript, a tudo o que existe dentro desta interface HTML. Por exemplo, se utilizar o HTML descrito e chamar o arquivo `js/scripts.js`, neste arquivo você terá acesso a todo esse HTML e a mais informações que o próprio navegador fornece naturalmente. Teste isso com esses comandos no arquivo `scripts.js`:

```
console.log(document);
console.log(document.head);
console.log(document.body);
```

Esse código exibirá todos os nós dentro da página no console:

Figura 7.1: Exibindo o DOM no console

Isso foi só para demonstrar como o JS tem acesso aos elementos do navegador. Cada item do HTML pode ser acessado e manipulado pelo JS. É assim que o JS consegue deixar as páginas de internet mais dinâmicas.

Acessando os nós

Temos algumas maneiras diferentes de acessar os elementos da interface. São elas:

document.querySelector

Esse seletor acessa o primeiro elemento que corresponde ao parâmetro de busca. Temos que usar a mesma sintaxe que usamos nos seletores de CSS. Se você tiver um trecho HTML assim:

```
<div class="conteudo" id="principal">
    <h1>Título</h1>
    <p>Um exemplo de parágrafo com um
        <a href="https://espaco.dev/">link aqui</a>.
    </p>
</div>
```

Você tem algumas formas de capturar esses elementos:

```
var conteudo = document.querySelector(".conteudo"); //captura por classe com .
console.log(conteudo);

var principal = document.querySelector("#principal"); //captura do mesmo elemento por id #
console.log(principal);
```

Você verá no console algo parecido com:

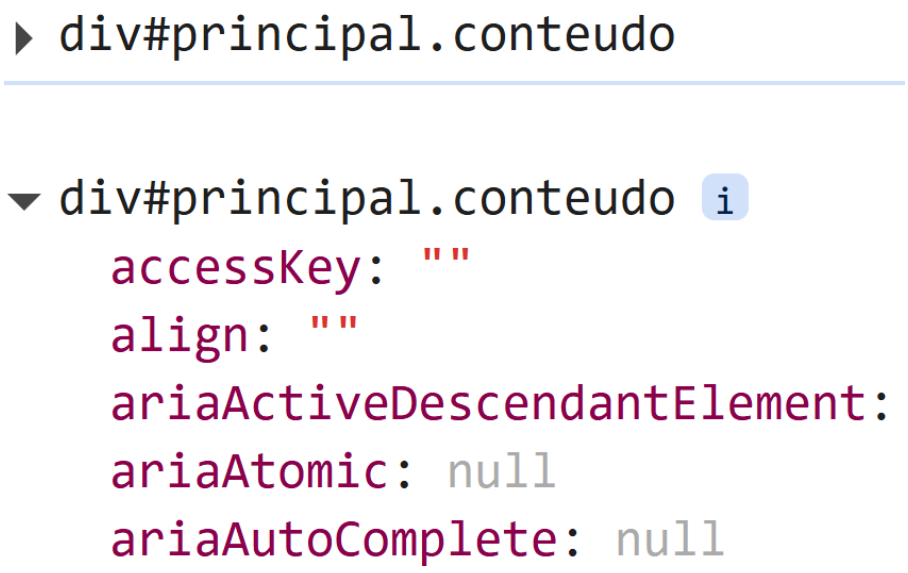


Figura 7.2: Seleção via querySelector

Observe na imagem que apareceu `div#principal.conteudo`. Isso significa que todo aquele elemento está dentro da variável. Isso mesmo. O JavaScript captura todo aquele elemento e coloca dentro de uma variável. Essa setinha da imagem está em destaque para mostrar que, ao clicar nela, serão exibidas diversas propriedades.

Isso quer dizer que, assim como o objeto com o qual trabalhamos no capítulo anterior, cada elemento da área de visualização possui diversas propriedades. Logo, se podemos alterar os valores das propriedades desses objetos do teste visto anteriormente, também podemos fazer o mesmo com os elementos da tela. E isso é feito da mesma forma que já praticamos. Vamos testar?

```
var conteudo = document.querySelector(".conteudo"); //Captura a classe
conteudo.style.backgroundColor = "gray"; //Altera a cor de fundo da classe
capturada
conteudo.style.color = "white"; //Altera a cor do texto da classe
capturada
conteudo.innerHTML = "Um novo texto aqui!"; //Altera o texto do conteúdo
da classe capturada
```

Agora sua tela estará assim:

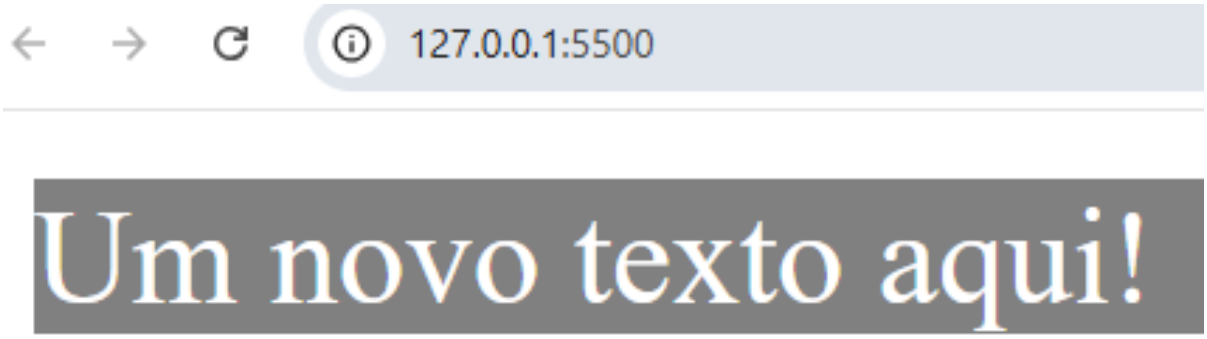


Figura 7.3: Conteúdo da tela alterada

Legal, né?! Fizemos nossa primeira alteração na tela.

document.getElementsByTagName

Esse código é usado para acessar um conjunto de elementos de toda uma estrutura HTML. Por ter essa característica menos específica, é uma forma menos utilizada. Um exemplo desse formato em um código HTML:

```
<div class="conteudo">
    <div>Conteudo 1</div>
    <div>Conteudo 2</div>
    <div>Conteudo 3</div>
</div>
```

Com uma seleção JS assim:

```
var conteudo = document.getElementsByTagName("div"); //Captura pela tag
<div>
console.log(conteudo[0]); //Captura toda a <div> pai com seus elementos
filhos
console.log(conteudo[1]); //Captura a 1ª <div> filho
console.log(conteudo[2]); //Captura a 2ª <div> filho
console.log(conteudo[3]); //Captura a 3ª <div> filho
```

Você percebeu que, por não ser uma seleção delimitada, os conteúdos estão entre `[]`, como se fossem `arrays`? Mas é isso

mesmo. Como essa seleção por tag não é específica, todos os elementos são atribuídos à variável em um único array.

document.getElementsByClassName

É usado para acessar o conjunto de elementos que usa a mesma classe de identificação. Também retorna um array com todos os elementos que possuem correspondência. Um exemplo seria como no código HTML:

```
<div>
    <div class="pets">Nerak</div>
    <div class="pets">Gil</div>
    <div class="pets">Toni</div>
    <div class="pets">Sushi</div>
</div>
```

Com uma seleção JS assim:

```
var conteudo = document.getElementsByClassName("pets"); //Captura pela classe
console.log(conteudo[0]); //Captura a <div> com conteúdo Nerak
console.log(conteudo[1]); //Captura a <div> com conteúdo Gil
console.log(conteudo[2]); //Captura a <div> com conteúdo Toni
console.log(conteudo[3]); //Captura a <div> com conteúdo Sushi
```

document.getElementById

Usado para acessar um único item da estrutura HTML. Este é o mais utilizado, pois permite capturar apenas um elemento, que será manipulado isoladamente. Imagine um código HTML assim:

```
<div>
    <div id="nome">Nerak</div>
    <div id="idade">10</div>
</div>
```

Com uma seleção JS assim:

```
var nome = document.getElementById("nome"); //Captura pelo ID nome
var idade = document.getElementById("idade"); //Captura pelo ID idade
nome.innerHTML = "Lucas"; //Altera o conteúdo da div com ID nome
idade.innerHTML = "35"; //Altera o conteúdo da div com ID idade
```

Dessa forma, é possível alterar vários itens da tela. Um a um, cada um no seu devido lugar.

Modificando os nós

Temos basicamente duas formas de modificar os textos dos nós:

- `innerText` : nos permite modificar o conteúdo textual do elemento;
- `innerHTML` : nos permite modificar o conteúdo do HTML interno do elemento.

Vejamos:

```
<div>
    <div id="titulo">Exemplo</div>
    <div id="texto">Exemplo</div>
</div>

var titulo = document.getElementById("titulo");
var texto = document.getElementById("texto");

titulo.innerText = "Meu titulo";
texto.innerHTML = "<strong>Um texto aqui</strong>";
```

Veja que com o `innerText` alteramos apenas o texto do elemento. Já com o `innerHTML` , podemos adicionar trechos de código HTML junto com o texto. O navegador interpreta essas novas tags e as renderiza corretamente na tela. Qual é melhor, ou qual devo usar? Depende do caso.

Criar novos elementos

Vimos como é possível alterar textos nos elementos e, agora, aprenderemos que também é possível adicionar novos elementos — ou seja, inserir códigos HTML dinamicamente por meio do JavaScript. Para isso, utilizamos a função `document.createElement()`, indicando o nome da tag HTML que representará o novo elemento.

Na sequência, é necessário adicionar esse nó como filho ao `body` ou a outro nó do documento atual, utilizando o comando `append()`.

```
var novo_paragrafo = document.createElement("p");
novo_paragrafo.innerHTML = "Este é um novo parágrafo com um <a
href='https://google.com'>link para o Google aqui</a>.";
document.body.append(novo_paragrafo);
```

Este código vai gerar um trecho HTML assim:

```
<p>
    Este é um novo parágrafo com um <a href='https://google.com'>link para
o Google aqui</a>.
</p>
```

E vai adicionar a sua página.

Remover elementos

De forma inversa, também podemos remover elementos da página. Basta selecionar o(s) elemento(s) de uma das formas que vimos e aplicar o `remove()`. Assim:

```
var subtitulo = document.getElementById("subtitulo");
subtitulo.remove();
```

Buscar dados dos campos

A maioria dos sites hoje em dia possui formulários para os usuários preencherem com informações de cadastro. O JavaScript faz o movimento de pegar as informações digitadas nos campos (`inputs`)

e também de colocar novas informações quando necessário. Imagine um formulário HTML de cadastro com esses dois campos:

```
<form>
    <input id="nome" type="text">
    <input id="idade" type="number">
</form>
```

No primeiro acesso, o usuário fornecerá as informações nesses campos e o JavaScript vai "pegá-las" para que sejam gravadas em um banco de dados. Para captar as informações no seu JS, basta buscar pela propriedade `value` do `input`:

```
var nome = document.getElementById("nome").value;
var idade = document.getElementById("idade").value;
```

A partir desse código, a informação digitada nos campos pode ser usada para montar arrays, objetos, enviar dados para um banco, encaminhar por e-mail ou outras finalidades.

Agora, em um novo acesso: o usuário já tem seu cadastro gravado e vai fazer um segundo acesso à plataforma. Nesse caso, é natural que a plataforma já exiba as informações para ele. O fluxo disso seria o JS buscar essas informações em um banco de dados e colocar no campo para que a informação fique visível para o usuário.

```
document.getElementById("nome").value = "Lucas";
document.getElementById("idade").value = "35";
```

Observe que é exatamente o fluxo inverso: antes, o JS obtém as informações dos campos; agora, insere-as.

Vamos criar um exemplo um pouco mais completo e robusto? Imagine que você precisa exibir uma lista de produtos que está em um array. Seu `index.html` está assim:

```
...
<body>
    <h1>Produtos</h1>
    <div id="conteudo">  <!-- Vamos adicionar os produtos neste nó -->
```

```
    </div>
    <script type="text/javascript" src="scripts.js"></script> <!-- Chamada
para o código JS -->
</body>
...
```

Aí você tem um array com os produtos, suas características e quer adicionar na tela. Olha como podemos fazer isso unindo as ferramentas de criação de elemento e laços de repetição:

```
var produtos = [
    { id: "1", nome: 'Camiseta', preco: 89.9, cor: "Azul" },
    { id: "2", nome: 'Calça', preco: 145.9, cor: "Azul" },
    { id: "3", nome: 'Boné', preco: 25.9, cor: "Vermelho" },
    { id: "4", nome: 'Tênis', preco: 129.9, cor: "Preto" }
];

for (var i = 0; i < produtos.length; i++) { //Vamos adicionar item por
item
    let item = document.createElement('div'); //Novo item é criado a cada
repetição

    //Geração do HTML para exibir o item formatado:
    item.innerHTML = "<h3>Item: " + produtos[i].nome + "</h3>" +
            "<b>Preço: " + produtos[i].preco + "</b>" +
            "<p>Cor: " + produtos[i].cor + "</p>" +
            "<span>Código de referência: " + produtos[i].id + "</span>" +
            "<hr />";
    let conteudo = document.getElementById('conteudo'); //Selecionamos o
nó pai
    conteudo.appendChild(item); //Adicionamos o item que foi criado ao nó
pai
}
```

O resultado da página será algo parecido com isso:

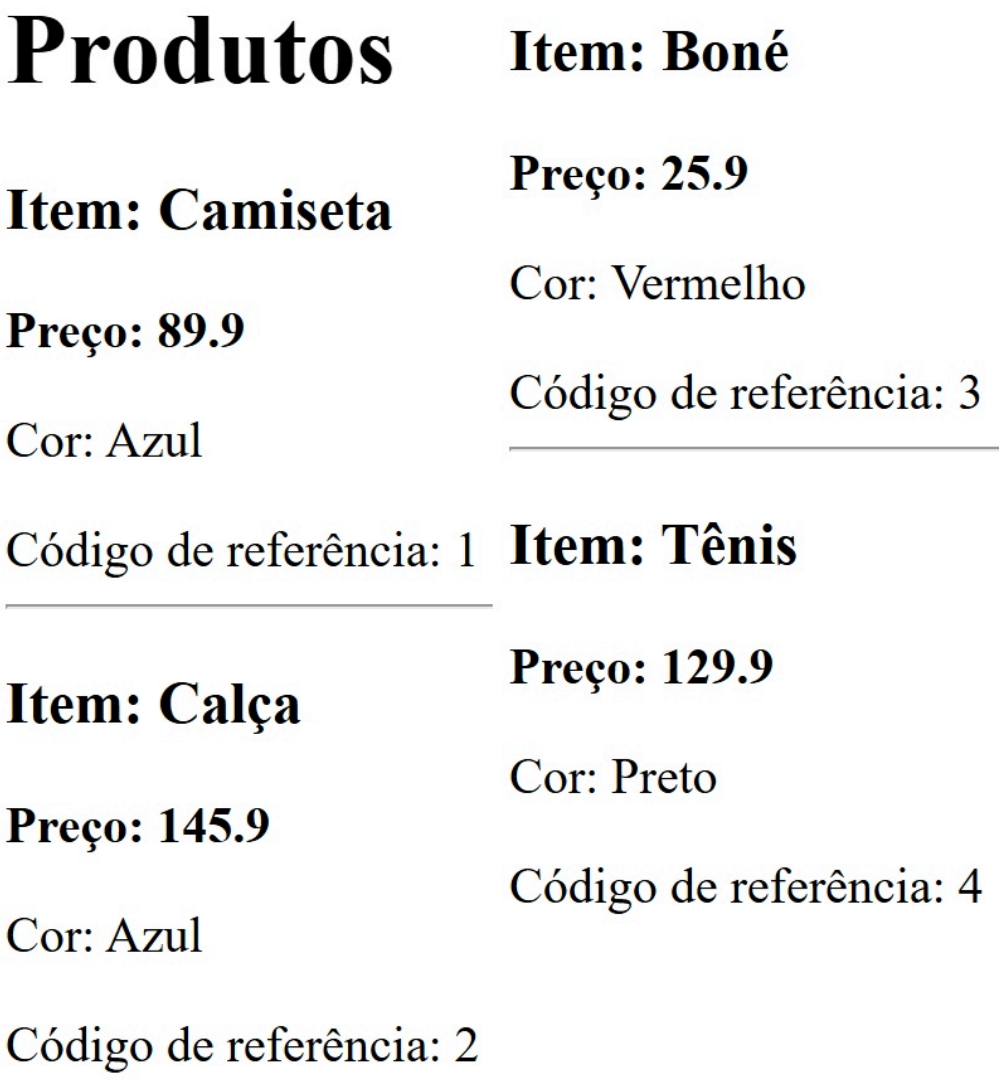


Figura 7.4: Resultado na página da lista criada

7.2 Eventos

Eventos são formas que temos, no JavaScript, de controlar as funcionalidades dos usuários e definir o comportamento da nossa plataforma.

Até agora, estávamos executando todo o código JS assim que a página era carregada. Com os eventos, podemos executar trechos

de códigos a partir de ações do usuário — como um clique em algum botão, a entrada em um campo ou o foco em determinado elemento. Existem diversos casos em que é possível acionar eventos e, consequentemente, executar códigos em JavaScript.

Alguns eventos possíveis e comuns:

- Eventos de teclado: `keyup`, `keydown`;
- Eventos de mouse: `click`, `double click`, `mouseover`, `mousedown`, `mouseup`;
- Eventos de alteração: quando é detectada uma mudança no valor do elemento, é gerado um evento `change`;
- Eventos de `input`: quando é digitada alguma coisa no campo;
- Eventos de `submit`: quando um formulário é enviado.

Há três formas de definir eventos no JavaScript. Veremos todas, mas fique à vontade para escolher a que mais deixa você confortável em sua organização de código.

Primeira forma - addEventListener

É um método de dois parâmetros que permite definir qual evento será "escutado" (por isso, o nome "Listener"). O primeiro parâmetro corresponde ao nome do evento; o segundo, à função que ele executará. Imagine um botão no seu código HTML e no seu código JS. É possível criar um evento assim:

```
<button id="boasvindas">Olá!</button>

let boasvindas = document.getElementById("boasvindas"); //Seleciona o botão
boasvindas.addEventListener("click", boasVindas); //Adiciona o evento de clique

function boasVindas(){ //Função que será acionada quando o botão for clicado
    alert("Bem-vindo Dev!");
}
```

Seu botão é criado na página HTML e o evento dele é criado no arquivo JS.

Segunda forma - Evento na propriedade

Podemos adicionar um evento diretamente na propriedade, acrescentando o prefixo **on**.

Também podemos usar funções anônimas para definir o comportamento do evento:

```
<button id="boasvindas">Olá!</button>

let boasvindas = document.getElementById("boasvindas"); //Seleciona o botão
boasvindas.onclick = () => {
    alert("Bem-vindo Dev!");
}
```

Nesse caso, adicionamos o evento na propriedade `click` com o prefixo **on**.

Terceira forma - Definir evento no HTML

Nesta forma, apontamos direto no HTML qual função será acionada no clique do botão. Aqui também precisamos do prefixo **on**. Assim:

```
<button id="boasvindas" onclick="boasVindas()">Olá!</button>

function boasVindas(){ //Função que será acionada quando o botão for clicado
    alert("Bem-vindo Dev!");
}
```

Concluímos o tópico de eventos. A ideia foi apresentar a noção de que o JS pode ser executado tanto no início do carregamento da página quanto em outros momentos, ao longo da interação que o

usuário faz em nossa página ou conforme informações que ele insere.

7.3 O formato JSON

Essa sigla vem de **JavaScript Object Notation**. O JSON é um formato de texto simples para representar grandes estruturas de dados. Sua sintaxe é muito similar à dos objetos que vimos em JS e, por isso, o JS trabalha de maneira bastante tranquila com esse formato.

JSON é comumente utilizado para enviar dados entre diferentes aplicações Web até mesmo de diferentes linguagens de programação. De forma simples, JSON é um texto com um formato específico. Quando houver a necessidade de enviar objetos JavaScript para um servidor, será necessário convertê-lo a esse formato. Para isso, temos dois métodos:

- `stringify` : recebe um objeto JS como parâmetro e retorna no formato de texto JSON;
- `parse` : recebe um texto JSON como parâmetro e retorna no formato de objeto JavaScript.

```
var pet = { name: 'Nerak', especie: 'Cachorro', idade: 10 }; //objeto

var petEmJSON = JSON.stringify(pet); //Convertendo objeto para JSON
var petObj = JSON.parse(petEmJSON); //Convertendo JSON para objeto

console.log(typeof petEmJSON); //string/texto
console.log(petEmJSON); //
{"name":"Nerak","especie":"Cachorro","idade":10}

console.log(typeof petObj); //objeto
console.log(petObj); // {"name":"Nerak","especie":"Cachorro","idade":10}
```

JSON e objetos parecem ser coisas iguais, mas são tipos diferentes. A boa notícia é que, sempre que se deparar com algo que parece ser um objeto, mas não é, com o JavaScript é fácil converter.

Também temos outros pontos a se considerar sobre o JSON.

- Os dados nesse formato podem ser armazenados em arquivos externos. Por exemplo, `dados.json` ;
- Esse formato contém apenas propriedades (diferente dos objetos, que podem carregar funções);
- Uma vírgula ou dois pontos mal posicionados podem produzir um JSON problemático;
- Uma diferença na sintaxe é que no código JavaScript as propriedades não ficam entre aspas; no JSON, somente as informações entre aspas podem ser utilizadas como propriedades.

7.4 APIs da Web nativas do JS

As APIs da Web são objetos nativos disponíveis para que o desenvolvedor possa usar e acessar recursos físicos do equipamento (do hardware) que o seu usuário está usando para ver seu site. Por exemplo, se sua plataforma gera um mapa com um ponto onde seu usuário está localizado, naturalmente esta plataforma precisa saber a localização do usuário e assim precisa dos dados do GPS do celular. O JavaScript é capaz de capturar as informações do GPS através das APIs da Web.

Como temos políticas de privacidade e de acesso à informação do usuário, nenhum site consegue automaticamente acessar um recurso físico do dispositivo do usuário. Há dois pré-requisitos:

- O componente tem que existir: para que seu aplicativo tenha acesso à localização do usuário, é necessário que exista o componente de GPS. Então, sua aplicação precisa estar

preparada para casos de acesso a partir de um aparelho mobile com GPS, mas também pode acontecer de ser de um notebook antigo que não tem esse componente e sua aplicação não terá informações de GPS;
- O usuário precisa conceder acesso: pelas políticas de privacidade e de acesso, seu JS só consegue obter alguns tipos de informações se o usuário permitir. Vamos supor que sua aplicação precisa acessar a câmera do usuário. Quando o comando para abrir a câmera do usuário for acionado, o aparelho dele exibirá uma mensagem perguntando se ele permite o acesso. A boa notícia é que essa pergunta ocorre apenas uma vez. Assim, uma vez que o usuário concede acesso ao seu site, essa solicitação não será exibida repetidamente, evitando incomodar seu cliente a todo momento.

Então, nos exemplos que veremos a seguir, só será possível testar se duas condições forem atendidas:

1. O dispositivo precisa ter o componente (como GPS ou câmera);
2. O usuário precisa conceder permissão de acesso.

Sem essas condições, a funcionalidade simplesmente não estará disponível.

As APIs da Web são muitas, com diversas funcionalidades legais e úteis. Vou exemplificar algumas a seguir, mas fique à vontade para ver a documentação completa e outros exemplos disponíveis no `MDN (Mozilla Developer Network) Web Docs` disponível em https://developer.mozilla.org/en-US/docs/Web/API.

Storage

O objeto Storage é uma Web API de armazenamento que permite armazenar dados de forma local no navegador do usuário, sem necessidade de conexão com um servidor. Com essa API, podemos preservar dados da aplicação no navegador do próprio cliente. O

navegador oferece dois tipos de *storage*: `localStorage` e `sessionStorage`.

Devemos usar esses armazenamentos para informações simples e rotineiras, pois qualquer pessoa com um pouco mais de habilidade técnica consegue visualizá-las. Então, muito cuidado! Evite armazenar senhas ou outros dados sensíveis com essa Web API.

As informações deste objeto são armazenadas no navegador por meio da aba **Aplicativo** da *Ferramenta do Desenvolvedor* (botão F12 no navegador):

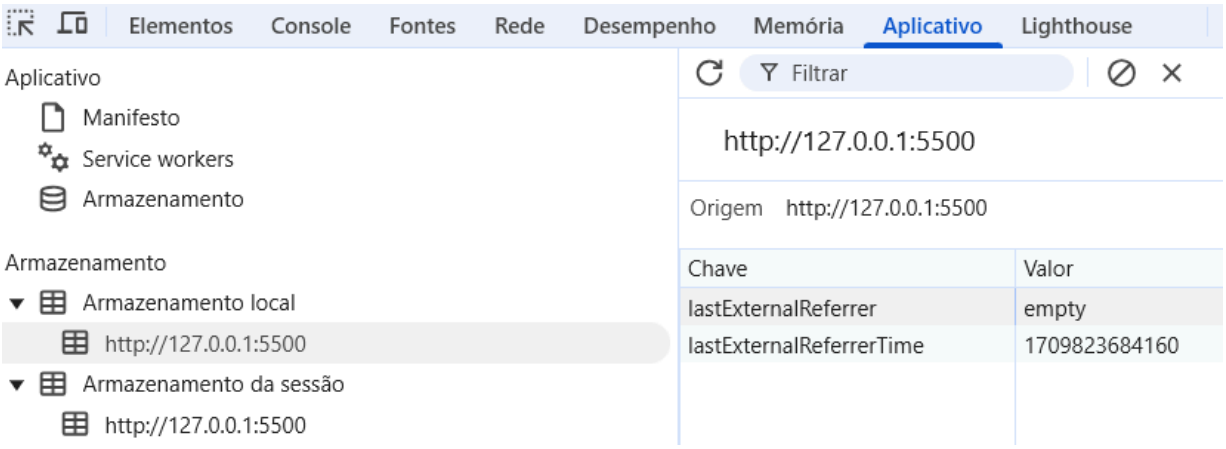


Figura 7.5: Aba Aplicativo da Ferramenta do Desenvolvedor

O navegador guarda os dados separando por links. Ou seja, ao mesmo tempo, o navegador pode armazenar diversas informações de diversos sites sem que uma informação interfira na outra. Então, em um primeiro acesso à sua plataforma, você consegue guardar algum dado no navegador do usuário e nenhum outro site conseguirá pegar essa informação.

localStorage

A informação é armazenada na forma de **chave-valor**. Isso também é semelhante aos objetos, mas aqui definimos chaves nas quais armazenamos nossos valores. As informações guardadas no `localStorage` são mantidas por tempo indeterminado — ou até que

sua própria plataforma as remova, ou que o usuário exclua os dados de navegação. Mesmo que o navegador ou até mesmo o computador seja reiniciado, as informações ainda estarão disponíveis.

Para armazenar uma informação é utilizado o `setItem` :

```
localStorage.setItem("usuario", "lucastauil");
localStorage.setItem("login_efetivado", true);
localStorage.setItem("data_login", new Date());
```

Esse código vai armazenar as informações de forma organizada, pelo link (que ainda é link de teste 127.0.0.1):

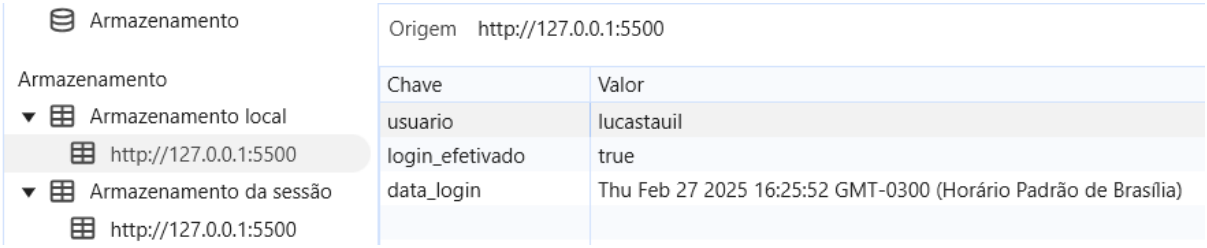


Figura 7.6: localStorage

E para buscar essas informações posteriormente:

```
var usuario_logado = localStorage.getItem("usuario");
var login_efetivado = localStorage.getItem("login_efetivado");
var data_do_login = localStorage.getItem("data_login");

console.log(usuario_logado); //lucastauil
console.log(login_efetivado); //true
console.log(data_do_login); //Thu Feb 27 2025 16:25:52 ...
```

Se em algum momento da execução você precisar deletar a(s) informação(ões) do navegador do usuário:

```
localStorage.removeItem("usuario", "lucastauil");
localStorage.removeItem("login_efetivado");
localStorage.removeItem("data_login");
```

sessionStorage

O funcionamento do `sessionStorage` é bem similar. A diferença é a duração da informação. Ela é armazenada somente até que o usuário feche a aba. Esse dado fica disponível naquele momento daquela aba atual. Se o usuário fechar o navegador e abrir novamente, a informação não estará mais lá. Assim, adicionamos:

```
sessionStorage.setItem("usuario", "lucastauil");
sessionStorage.setItem("email", "luctauil@gmail.com");
```

Podemos buscar esses dados assim:

```
var usuario_local = sessionStorage.getItem("usuario");
var email = sessionStorage.getItem("email");

console.log(usuario_logado); //lucastauil
console.log(email); // luctauil@gmail.com
```

E para remover:

```
sessionStorage.removeItem("usuario", "lucastauil");
sessionStorage.removeItem("email");
```

Bateria

Adicionei essa Web API aqui pois ela é uma das mais simples de usar. Sua finalidade é fornecer ao usuário como está o status da bateria do dispositivo. Rotineiramente, esse conteúdo não costuma ser exigido pelo usuário, mas, em alguns projetos específicos, pode ser importante avisá-lo que sua aplicação consome mais bateria e perguntar se realmente deseja continuar.

Imagine que você está desenvolvendo um aplicativo que é uma simples lanterna. Seria de bom tom informar ao usuário que a bateria dele está baixa e sua aplicação faz uso do flash do celular.

Veja como usamos essa Web API:

```
function CapturarStatusBateria() {
    navigator.getBattery()
        .then(function(bateria) {
```

```
            var nivel = bateria.level * 100; // Obtém o nível da bateria
em porcentagem
            var paragrafo = document.getElementById("status");
            paragrafo.innerHTML = "Status da bateria: " + nivel + "%";
        }).catch(function(error) {
            paragrafo.innerHTML = "Não foi possível obter informações da
bateria";
        });
}

//Adicionar o evento a um botão qualquer da sua tela
var capturar = document.getElementById("capturar");
capturar.addEventListener("click", CapturarStatusBateria);
```

Este trecho de código considera que, no seu HTML, existe um botão (`<button>`) com o `id` "capturar", e que esse botão acione a função `CapturarStatusBateria` . Tudo que está dentro dessa função é formatado pela API — mas somos nós que decidimos como essas informações serão exibidas na tela. No exemplo, optamos por mostrar o resultado dentro de um parágrafo (`<p>`) com o `id` "status", que também precisa estar presente no HTML.

Observe que o texto que vai para esse parágrafo é adicionado em 2 momentos: o primeiro, avisando o status da bateria com uma variável `nivel` ; o segundo, comunicando que não foi possível obter informações da bateria. Esse segundo trecho da Web API já está preparado para o erro de não haver uma bateria no dispositivo do usuário. Se seu usuário acessar de um notebook, tablet ou de um smartphone, a informação da bateria será exibida. Mas pode acontecer do seu usuário acessar de um computador desktop que não tem uma bateria e, para evitar um erro estranho, é só informar que "Não foi possível obter informações da bateria" ou "Bateria não encontrada.". É só alterar o texto conforme você deseja exibir ao usuário.

Geolocalização

Já comentamos sobre o uso do GPS na seção anterior, mas agora vamos conhecer mais de perto a API específica de geolocalização. Trata-se de uma das Web APIs mais práticas, pois permite identificar a posição geográfica atual do usuário com base nos recursos disponíveis no dispositivo.

Como essa informação é sensível do ponto de vista da privacidade, o navegador só permite o acesso se o usuário conceder permissão.

Veja o exemplo:

```
if (navigator.geolocation) {
    navigator.geolocation.getCurrentPosition(function (position) {
            var latitude = position.coords.latitude;
            var longitude = position.coords.longitude;
            var precisao = position.coords.accuracy;
            console.log("latitude: " + latitude);
            console.log("longitude: " + longitude);
            console.log("precisao: " + precisao);

            var informacoes = document.getElementById("informacoes");
            informacoes.innerHTML = "Latitude: " + latitude + "
<br>Longitude: " + longitude + "<br>Precisão: " + precisao;
    });
}
else{
    alert("Não foi possível acessar as informações de localização.");
}
```

Quando a pessoa acessar seu site, aparecerá um pedido de autorização parecido com isso:

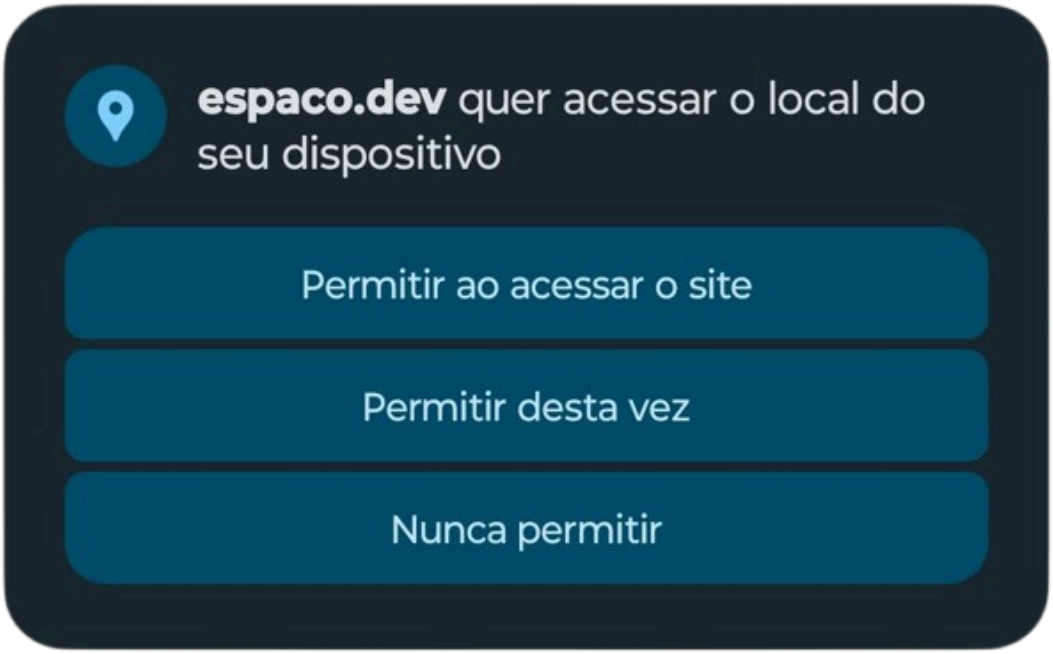


Figura 7.7: Pedido de permissão a geolocalização

Com esse simples trecho de código (e a permissão do usuário), seu site pode:

- exibir a localização atual do visitante no mapa;
- calcular distâncias entre o usuário e um ponto específico;
- destacar áreas de interesse com base na posição do usuário...

As possibilidades são inúmeras.

As três coordenadas (latitude, longitude e precisão) vão para as variáveis, e você pode utilizá-las da maneira que desejar no seu site ou aplicativo.

É bom observar a variável de precisão, pois os dispositivos usam diversos recursos para conseguir **estimar** a localização do usuário, como o GPS (informação mais confiável), **Wi-Fi Position System**, triangulação de torres de celular, sensores internos (acelerômetro e giroscópio) etc.

A precisão é exibida de forma numérica e esse número representa metros. Se sua página for acessada por um smartphone, é muito provável que a informação seja exibida com menos de 5 metros, dependendo da precisão do GPS do smartphone. Trata-se de uma boa precisão.

Mas, se sua página for acessada por um notebook, é provável que a única informação que o JavaScript tenha para estimar a localização seja a Wi-Fi do roteador mais próximo. O valor desta variável pode chegar a 2.000, indicando que o dispositivo está em um raio de 2 quilômetros e, dependendo da aplicação, isso é uma péssima precisão. Então, fique atento e prepare seu programa para valores assim. Pode ser que você tenha que descartar o dado da localização.

Essas não são as únicas Web APIs existentes. Trouxe algumas apenas para aguçar o seu desejo de se aprofundar em JS. Vamos manter o foco nos pontos necessários para iniciarmos o trabalho com Dashboards. Deixo aqui toda a documentação para você ver as demais APIs da Web existentes no JS: https://developer.mozilla.org/pt-BR/docs/Web.

7.5 Assincronia

Estamos em uma parte do livro realmente emocionante. As coisas incríveis do JavaScript começaram a aparecer e, até o fim do material, não há perspectiva de que isso diminua. O modelo de assincronia é mais um daqueles conceitos que podem mudar a forma como você enxerga a programação.

Até aqui, todo o código que escrevemos foi executado de forma linear, uma linha depois da outra, como se estivesse em uma fila. Uma instrução é executada e somente após seu resultado a segunda é iniciada. Esse modelo é chamado de **execução síncrona**. Isso

significa que uma tarefa só começa depois que a anterior termina. Algo como:

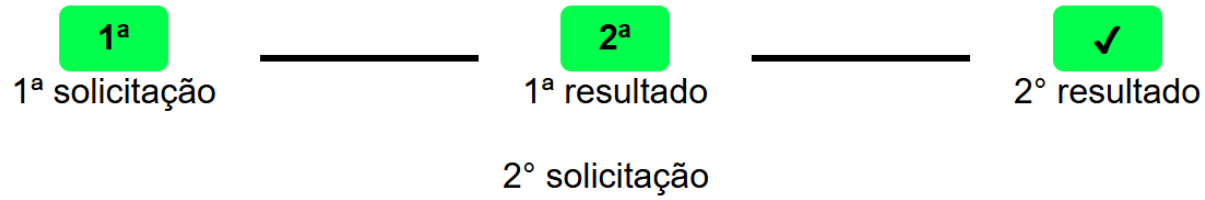


Figura 7.8: Programação síncrona

O que vamos aprender neste tópico é que o JavaScript também permite programação **assíncrona**. Esse modelo permite que múltiplas ações ocorram simultaneamente. Ao começar uma ação, o programa continua executando o restante do código. Quando a ação anterior termina, o programa é notificado e pode acessar o resultado correspondente. É como disparar várias ações ao mesmo tempo, visualmente parecido com isto:

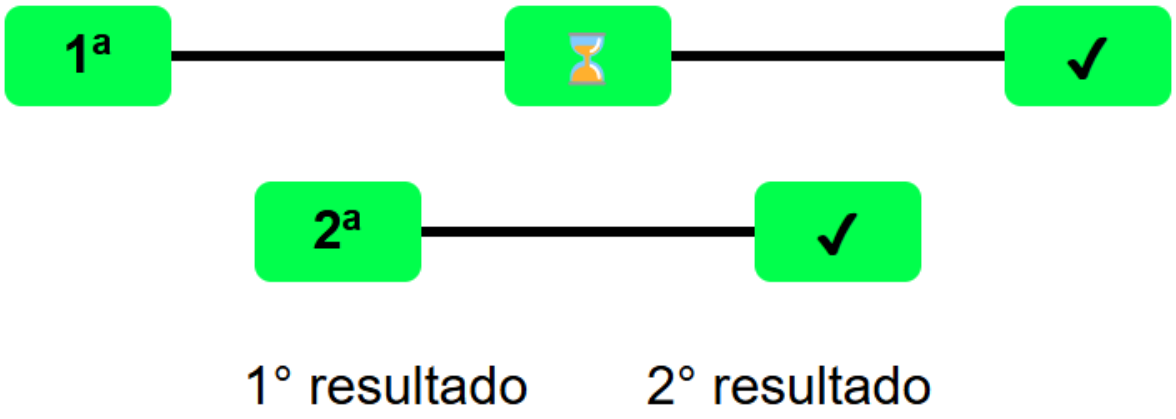


Figura 7.9: Programação assíncrona

De forma resumida:

- Modelo síncrono: a próxima solicitação é iniciada somente quando a primeira terminar. Temos o resultado de todas as ações somente após a última terminar.

- Modelo assíncrono: duas (ou mais) solicitações podem ser executadas em paralelo, só precisamos sincronizar os resultados ao finalizar.

Principal vantagem deste novo modelo: facilita a manipulação de sistemas que realizam múltiplas tarefas, e o carregamento de algumas informações não atrasa o restante da tela.

Principal risco: pode dificultar a compreensão de programas que seguem uma única linha de execução.

Então, basicamente, temos que aprender quando e como sincronizar os resultados com a execução atual quando houver programação assíncrona. Vamos fazer isso vendo alguns facilitadores.

Função setTimeout()

Essa função nativa permite realizar ações assincronamente. Ela recebe dois parâmetros, uma função que chamamos de **callback** e um valor numérico que representa uma quantidade de milissegundos. Isso significa que a função que passamos como primeiro parâmetro é executada após o tempo decorrido do valor do segundo parâmetro.

Vamos aplicar um exemplo mais elaborado dessa função para observar o comportamento assíncrono:

```
console.log("Início do script!");
setTimeout(() => {
    console.log("Meio do script!");
}, 5000);
console.log("Final do script!");
```

Observe a ordem dos códigos de instrução e observe como exibirá no console do navegador:

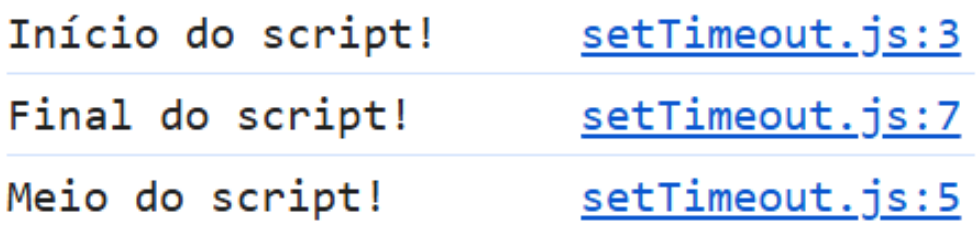


Figura 7.10: Resultado setTimeout

Olha que interessante. A instrução que está no meio do código foi exibida por último. O JavaScript iniciou a execução desse código, viu que havia um código assíncrono no meio do caminho que aguarda 5 segundos, continuou sua execução e, por fim, depois dos 5 segundos voltou para executar.

Função setInterval()

É muito parecido com `setTimeout`, mas o parâmetro do tempo representa um intervalo até uma nova execução. Observe um exemplo interessante:

```
setInterval(() => {
    var d = new Date();
    var relogio = document.getElementById("paragrafo");
    relogio.innerHTML = d.toLocaleString();
}, 1000);
```

Nesse código, o `setInterval` tem no seu primeiro parâmetro uma instrução que pega a data atual, depois pega um componente de `paragrafo` e então insere o horário atual neste componente que está na tela.

O segundo parâmetro determina um intervalo de 1000 milésimos para que o trecho seja novamente executado. Basicamente, o horário atual vai ser atualizado a cada um segundo. Tudo isso junto gera um relógio em tela. Resumindo, o `setInterval` recebe um bloco de código que aguarda um tempo definido e fica se executando.

Mas o `setInterval` fica executando para sempre? Quando há necessidade de parar a execução do `setInterval` antes do encerramento da página, podemos usar o `clearInterval`. Podemos armazená-lo em uma variável e definir uma regra para passar essa referência à função `clearInterval`.

```
var intervalo = setInterval(() => {
    var d = new Date();
    var relogio = document.getElementById("paragrafo");
    relogio.innerHTML = d.toLocaleString();
    if(d.getMinutes() == 0) {
        clearInterval(intervalo);
    }
}, 1000);
```

O código descrito é o mesmo do anterior com a diferença que todo o `setInterval` é colocado dentro de uma variável e dentro da função existe uma estrutura `if` que define a limpeza desta variável quando o horário atual tiver no minuto zero.

Promessas

Ainda nessa *vibe* de assincronia, temos que aprender a usar as promessas (ou *Promises*). As promessas são objetos em JS que representam um evento futuro que tem uma ação assíncrona que pode se completar em algum momento, produzir um valor e avisar quando isso acontecer. Uma promessa tem 3 estados possíveis:

- `pending` (pendente): quando a ação assíncrona ainda não foi finalizada;
- `fulfilled` (resolvida): quando a ação foi concluída com sucesso;
- `rejected` (rejeitada): quando ocorreu um erro a execução.

Inicialmente, a promessa retorna com estado `pending`, pois fica entendido que o valor a ser gerado ainda não está definido. Quando o retorno da promessa é definido quer dizer que ou a função se resolve (estado `fulfilled`) ou é rejeitada (estado `rejected`).

Por que isso é útil? Para entender melhor, imagine que estamos desenvolvendo um Dashboard empresarial com várias informações em uma única página:

- Um gráfico de linha com o fluxo de vendas do mês;
- Um gráfico de barras comparando as vendas dos últimos 12 meses;
- Um quadro de preços dos produtos mais usados na empresa;
- Um quadro de previsão do tempo.

Cada um desses elementos precisa buscar dados de diferentes fontes. Se a aplicação rodar de forma sequencial, o carregamento pode travar caso um dos servidores esteja fora do ar. Isso significa que nenhuma informação será exibida até que todos os dados sejam carregados.

Agora, se utilizarmos Promises, conseguimos disparar essas 4 requisições em paralelo. Se um banco de dados estiver indisponível, apenas um gráfico será afetado, enquanto os outros continuam funcionando normalmente. Isso torna a aplicação mais eficiente e resiliente. Vejamos o mecanismo das promessas:

```
function verificarNumero(numero) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            if (numero % 2 === 0) {
                resolve(`O número ${numero} é par!`);
            } else {
                reject(`O número ${numero} é ímpar!`);
            }
        }, 3000); // Simula uma operação assíncrona de 3 segundos
    });
}

console.log("Início da verificação...");

verificarNumero(4)
    .then(resultado => console.log("Sucesso: " + resultado)) //Caso de
```

```
sucesso
    .catch(erro => console.log("Erro: " + erro)) //Caso de erro

console.log("Aguardando resultado...");
```

O que está acontecendo nesse código:

- A função `verificarNumero(numero)` retorna uma `Promise` ;
- Dentro dela, usamos `setTimeout()` para simular uma operação que demora 3 segundos;
- Se o número for par, chamamos `resolve()` , indicando um sucesso (`fulfilled`);
- Se for ímpar, chamamos `reject()` , indicando uma falha (`rejected`);
- A função retorna imediatamente uma Promise pendente, permitindo que o código continue rodando sem esperar;
- O `.then()` captura o sucesso e o `.catch()` captura os erros.

A saída esperada no console é:

```
"Início da verificação...
Aguardando resultado...
Sucesso: O número 4 é par! (após 3 segundos)"
```

Esse exemplo ilustra bem como uma Promise representa um valor futuro. O `new Promise` recebe duas funções como parâmetros:

- Uma para ser executada em caso de sucesso (`resolve`), que podemos associar ao `.then()` ;
- Outra para ser executada em caso de falha (`reject`), que será tratada no `.catch()` .

No caso do nosso Dashboard, cada gráfico poderia ser carregado dentro de uma Promise, garantindo que a aplicação continue funcionando mesmo se alguma requisição falhar. Isso melhora performance, experiência do usuário e robustez do sistema.

7.6 Fetch

Para falar desta ferramenta precisamos fazer uma breve contextualização. Por volta dos anos 2000, os programadores de página web começaram a ter constantes desafios para receber e enviar informações sem afetar o estado atual da página, sem precisar de um novo carregamento total ou atualização completa da página. Para qualquer atualização de informação na página, por menor que ela fosse, era necessário carregar toda a página novamente.

AJAX

Para solucionar isso, surgiu o AJAX (sigla de *JavaScript Assíncrono e XML*) que foi um conjunto de técnicas desenvolvidas que permitem que as aplicações funcionem de forma assíncrona, podendo carregar e processar informações em segundo plano. A partir disso, qualquer site web que utilizasse AJAX podia enviar e receber dados sem precisar carregar novamente toda a página. Dessa forma foi possível evitar interrupções realizadas pelo usuário, adicionar mais dinamismo e características essenciais dos softwares modernos que temos até hoje.

Essas atualizações de página (e até mesmo o carregamento inicial) utilizam um mecanismo de solicitação de dados chamado HTTP (*Hypertext Transfer Protocol*). Quando fazemos o pedido de um site no nosso navegador, ele faz uma solicitação (requisição) HTTP para algum servidor. Esse servidor vai receber, processar esse pedido e devolver uma resposta com as informações que utilizamos no site.

Todo site que você acessa é uma solicitação ou requisição a um outro computador remoto (chamado servidor) que fornece dados para o seu navegador. As solicitações possuem estados que respondem através de códigos:

- Códigos que começam com 2 (geralmente 200): representam solicitações de sucesso. Quando a requisição retorna com esse

código, seu JS com certeza recebeu informações de forma correta e sua página será carregada normalmente;
- Códigos que começam com 4: significa que houve algum erro com a solicitação e geralmente no lado do usuário. O famoso erro 404 indica que o recurso não foi encontrado. Se você tiver um problema de internet, seu navegador não conseguirá acessar o servidor e, assim, ocorrerá um erro de recurso não encontrado;
- Códigos que começam com 5 significa que houve erro com o servidor. Não é problema no lado do usuário que está acessando. Foi o servidor que não conseguiu retornar corretamente as informações para o navegador do solicitante.

APIs

Entendendo esse procedimento de busca de informações do cliente e respostas de um servidor na internet, podemos introduzir um outro conceito contextualizador: a tal da API. Uma API (*Application Programming Interfaces*) é uma aplicação que fica disponível na internet pronta para devolver informações para os clientes em um formato específico. Elas permitem que qualquer sistema, seja em JavaScript ou outra linguagem, acesse e utilize essas informações de forma prática.

Vamos imaginar que queremos desenvolver um site que fornece previsão do tempo. Antes de fazer o site, teríamos que ter diversos sensores e termômetros em todas as regiões que queremos e depois montar um banco de dados organizados. É simples fazer isso? É barato? Não. Mas existem grandes empresas que fazem e, se elas quiserem, podem compartilhar essas informações via API e qualquer pessoa pode usar em sua aplicação. Hoje em dia temos diversas APIs gratuitas de previsão do tempo. Existem diversas APIs de diversos assuntos gratuitos que conseguimos buscar informações e montar sites incríveis baseado em informações que não temos, mas que outras empresas têm e nos permitem utilizar através de APIs.

Para entender melhor como isso funciona, vamos usar uma API bem simples, que serve justamente para praticar esse conceito. Existe uma API gratuita que fornece todos os dados de Pokémons. Para aprender a usar todos os detalhes dessa API, acesse o site oficial no link: https://pokeapi.co. Mas vamos ver um link direto já de um pokémon: https://pokeapi.co/api/v2/pokemon/pikachu. Se você acessar esse link verá diversas informações de forma confusa. Algo como:


```
{"abilities":[{"ability":
{"name":"static","url":"https://pokeapi.co/api/v2/ability/9/"},"is_hidden":false,"slot":1},
{"ability":{"name":"lightning-
rod","url":"https://pokeapi.co/api/v2/ability/31/"},"is_hidden":true,"slot":3}],"base_experien
ce":112,"cries":
{"latest":"https://raw.githubusercontent.com/PokeAPI/cries/main/cries/pokemon/latest/25.ogg","
legacy":"https://raw.githubusercontent.com/PokeAPI/cries/main/cries/pokemon/legacy/25.ogg"},"f
orms":[{"name":"pikachu","url":"https://pokeapi.co/api/v2/pokemon-form/25/"}],"game_indices":
[{"game_index":84,"version":{"name":"red","url":"https://pokeapi.co/api/v2/version/1/"}},
{"game_index":84,"version":{"name":"blue","url":"https://pokeapi.co/api/v2/version/2/"}},
{"game_index":84,"version":{"name":"yellow","url":"https://pokeapi.co/api/v2/version/3/"}},
{"game_index":25,"version":{"name":"gold","url":"https://pokeapi.co/api/v2/version/4/"}},
{"game_index":25,"version":{"name":"silver","url":"https://pokeapi.co/api/v2/version/5/"}},
{"game_index":25,"version":{"name":"crystal","url":"https://pokeapi.co/api/v2/version/6/"}},
{"game_index":25,"version":{"name":"ruby","url":"https://pokeapi.co/api/v2/version/7/"}},
{"game_index":25,"version":{"name":"sapphire","url":"https://pokeapi.co/api/v2/version/8/"}},
{"game_index":25,"version":{"name":"emerald","url":"https://pokeapi.co/api/v2/version/9/"}},
{"game_index":25,"version":{"name":"firered","url":"https://pokeapi.co/api/v2/version/10/"}},
{"game_index":25,"version":
{"name":"leafgreen","url":"https://pokeapi.co/api/v2/version/11/"}},
{"game_index":25,"version":{"name":"diamond","url":"https://pokeapi.co/api/v2/version/12/"}},
```


Figura 7.11: Resposta da PokéAPI

Mas se você olhar melhor verá que esse monte de informações estão em no formato JSON. Isso mesmo! E como vimos anteriormente, o JS integra perfeitamente com JSON. Veja que as APIs fornecem diversas informações. Analisando esse link, ele tem fotos do Pikachu, tem habilidades, poderes, fatores de raridade etc. Um monte de informações que existiam no jogo.

As informações estão lá, mas usamos apenas as que são interessantes para nós. Assim, finalmente podemos ver a função `fetch` em ação. Essa função recebe um link que indica a origem da informação, que depois pode ser aplicada do jeito que quisermos. No nosso exemplo aqui, só vamos usar imagem e o nome.

```
fetch("https://pokeapi.co/api/v2/pokemon/pikachu")
    .then(response => response.json())
    .then(data => {
        console.log(data); //Apenas para vermos o objeto inteiro no
console

        document.getElementById("nome").textContent = data.name;
        document.getElementById("imagem").src =
data.sprites.front_default;
    })
    .catch(error => console.error("Erro ao buscar Pokémon:", error));
```

Dentro do `fetch` informamos o link para o JS buscar os dados. Depois, temos dois `.then` que executam sequencialmente a formatação do JSON e a conversão das informações buscadas em um objeto `data` (inglês para "dados"). Veja que o objeto `data` possui a característica `name` e também `sprites`. Essas características são todas que foram convertidas do JSON. Cada API retornará suas próprias características.

O comando do `console.log` deixamos para visualizar o objeto completo:

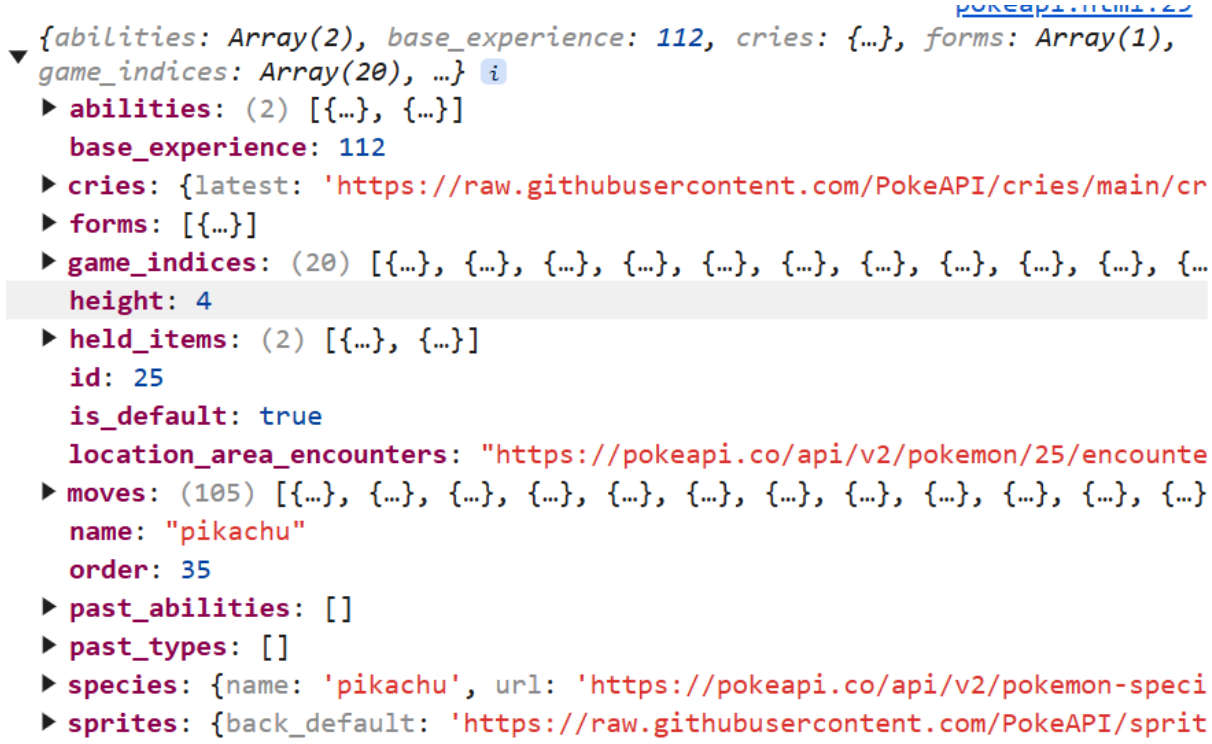


Figura 7.12: Objeto completo buscado pelo fetch disponível em nossa página

Async e Await

Falamos sobre o `fetch` e de como ele utiliza `Promises` para buscar informações. Agora, vamos considerar uma outra forma de lidar com operações assíncronas que deixam o código **mais limpo e legível**. São as palavras-chave `async` e `await`.

Para entender melhor, precisamos lembrar que toda **Promise demora um tempo para ser resolvida**. No nosso exemplo da PokéAPI, tivemos que usar `.then()` para capturar o resultado da requisição e `.catch()` para lidar com possíveis erros. Mas imagine que queremos lidar com múltiplas requisições seguidas ou processar os dados de maneira mais intuitiva.

Aqui entra o `async/await`, que basicamente permite esperar a conclusão de uma `Promise` sem precisar de `.then()`. Isso faz com que nosso código pareça mais sequencial e fácil de entender.

Como funciona? Para transformar uma função em assíncrona, basta adicionar `async` antes dela. E para aguardar o resultado de uma `Promise` dentro dessa função, usamos `await`. Vamos reescrever nosso exemplo da PokéAPI, agora usando `async/await`:

```
async function buscarPokemon() {
    try {
        let response = await
fetch("https://pokeapi.co/api/v2/pokemon/pikachu"); // Aguarda a resposta
        let data = await response.json(); // Aguarda a conversão para JSON

        console.log(data); // Apenas para visualizarmos o objeto completo
no console

        document.getElementById("nome").textContent = data.name;
        document.getElementById("imagem").src =
data.sprites.front_default;
    } catch (error) {
        console.error("Erro ao buscar Pokémon:", error);
    }
}

buscarPokemon(); // Chamamos a função para buscar os dados
```

O que mudou em relação ao `.then()`:

- A função `buscarPokemon()` agora tem `async` antes do nome, indicando que ela lida com operações assíncronas;
- Em vez de `.then()`, usamos `await` para dizer ao JavaScript: "Espere a resposta do fetch para continuar" e "Espere a conversão para JSON para seguir adiante".
- Usamos `try...catch` para capturar possíveis erros (antes, usávamos `.catch()`).

Assim o código fica mais fácil de ler e entender, com menos `.then()` encadeados e mais intuitivo para quem está acostumado com código

síncrono.

Agora que entendemos `async` e `await`, podemos criar requisições assíncronas de uma maneira muito mais organizada. Em qualquer projeto que trabalhe com APIs, esse conhecimento será essencial.

Eu sei que você deve estar pensando que sempre há uma forma mais fácil de fazer algo e o escritor deste livro só apresenta isso depois. É isso aí! Eu quero deixar você pronto para tudo, para entender códigos bons e ruins, organizados e bagunçados. Mas com certeza com a prática você vai se habituar a usar os códigos e procedimentos eficientes.

7.7 Bibliotecas

Bibliotecas são conjuntos de códigos prontos, criados por terceiros, que ajudam a resolver tarefas comuns em nossos projetos. Elas estão presentes em várias linguagens de programação e o JavaScript também tem as suas.

Podemos pensar em uma biblioteca como **um conjunto de ferramentas especializadas**, que podemos usar no nosso site ou aplicação. Isso acelera o desenvolvimento e evita que a gente tenha que "reinventar a roda".

> **IMPORTANTE:** no Capítulo 1, falamos sobre frameworks. A diferença é que **bibliotecas oferecem ferramentas específicas que usamos quando precisamos**, enquanto frameworks geralmente definem toda a estrutura do projeto e como você deve trabalhar com ela. Ou seja, com a biblioteca você está no controle. Com o framework, é ele que dita as regras.

Estão disponíveis milhares de bibliotecas JavaScript na internet. As mais conhecidas têm excelente documentação e são amplamente utilizadas pela comunidade. Quando for usar alguma, **sempre acesse o site oficial** e leia a documentação. Geralmente, em poucos passos, você consegue começar a usar.

E se não houver um site e uma boa documentação da biblioteca que eu quero usar? É melhor tomar cuidado com essa biblioteca e talvez procurar outra, pois as bibliotecas confiáveis sempre contam com uma boa documentação e ampla aceitação pela comunidade na internet.

Exemplos de problemas que essas bibliotecas podem resolver:

- Validação de dados: você pode ter auxílio para enviar ou buscar informações da sua página;
- Interfaces visuais: podem existir componentes prontos com alertas mais bonitos que você não precisará se preocupar com a construção deste alerta;
- Manipular datas: formatação na exibição das datas, calcular datas etc.

Como usar uma biblioteca?

Você pode encontrar expressões como: "implementar no projeto", "instalar", "incorporar ao site", "integrar ao sistema". Todos esses termos significam a mesma coisa: **usar o código da biblioteca no seu projeto**. Elas são adicionadas como qualquer outro script JavaScript no HTML. Existem dois jeitos principais:

1. Usando um arquivo baixado: Você faz o *download* da biblioteca e adiciona na pasta do seu projeto e chama assim:

   ```
   <script src="arquivo_baixado.js"></script>
   ```

2. Você usa o link fornecido na documentação oficial da biblioteca, como neste exemplo:

```
<script src="https://cdn.biblioteca.com/versao/biblioteca.min.js">
</script>
```

Os dois formatos são válidos e funcionarão perfeitamente. A diferença é que, no primeiro, você terá no seu projeto o(s) arquivo(s) da biblioteca e seu projeto não dependerá da internet para rodar e, no segundo, é exatamente o inverso, você não terá os arquivos no seu projeto, mas ele precisará de acesso à internet para buscar sempre essa biblioteca.

Qual é o melhor formato para se usar? Sinceramente, as diferenças são pequenas considerando que quase todas as bibliotecas são pequenas e quase nunca será necessário rodar a aplicação offline. Escolha uma delas conforme achar mais fácil na documentação.

Aliás, aproveito esse momento para comentar que, geralmente, as bibliotecas fornecem acesso a arquivos minificados (ou *minified*) que são arquivos cujo conteúdo JS está em uma única linha. Parece um arquivo com códigos compilados, juntos e compactados. Isso porque os desenvolvedores de biblioteca têm a preocupação de fornecê-las em arquivos de tamanho pequeno. A remoção desses espaços reduz significativamente o tamanho das bibliotecas.

Com esses conceitos em mãos, vamos conhecer algumas bibliotecas.

SweetAlert

Essa biblioteca nos permite criar alertas personalizados, muito atraentes, simples de usar e personalizáveis. Lembra que usamos vários daquele tradicional `alert()` que gera isto?

Figura 7.13: Alert padrão do navegador

Pois bem, vamos usar o `SweetAlert`, que é muito mais atraente. E como ele é uma biblioteca, não precisamos desenvolver nada. É só instalar e usar.

O site com a documentação é o https://sweetalert2.github.io/ ou apenas digite Sweet Alert no Google, que aparecerá. No menu principal do site você encontrará **Installation** e lá está como instalar e a forma de usar.

```
<body>
    <h1>Sweet Alert</h1>
    <button onclick="Swal.fire('Olá Desenvolvedor!')">Clique aqui</button>
    <script src="https://cdn.jsdelivr.net/npm/sweetalert2@11"></script>
</body>
```

A instalação está feita através do link da CDN e o uso está direto no botão. Mas o uso não precisa ser neste formato, você pode colocá-lo dentro de uma função JS, pode colocá-lo em funções assíncronas, ou talvez depois de uma rotina de gravar informações. Enfim, você só precisa colocar a linha de instalação do HTML e a forma de uso depois é livre. Neste exemplo você terá um resultado assim:

Figura 7.14: Exemplo do Sweet Alert

Percebe a diferença? Não precisamos programar nada deste alerta, ele veio pronto para nós. Com todo o efeito visual, o charme da mensagem e do botão, o efeito de escurecer o restante da tela para ter foco maior no alerta... Não programamos nada disso, só usamos. E ainda é possível personalizar cores de botão, mensagens etc.

Ainda com esse mesmo HTML, vamos simular que esse botão chama a função `GravarInformacoes()` e testar com maior personalização:

```
function GravarInformacoes(){
    /* Rotina que grava informações e depois um alerta de sucesso  */
    Swal.fire({
        title: "Sucesso!",
        icon: "success",
        text: "Informações gravadas com sucesso!",
        draggable: true
      });
}
```

Você verá o alerta:

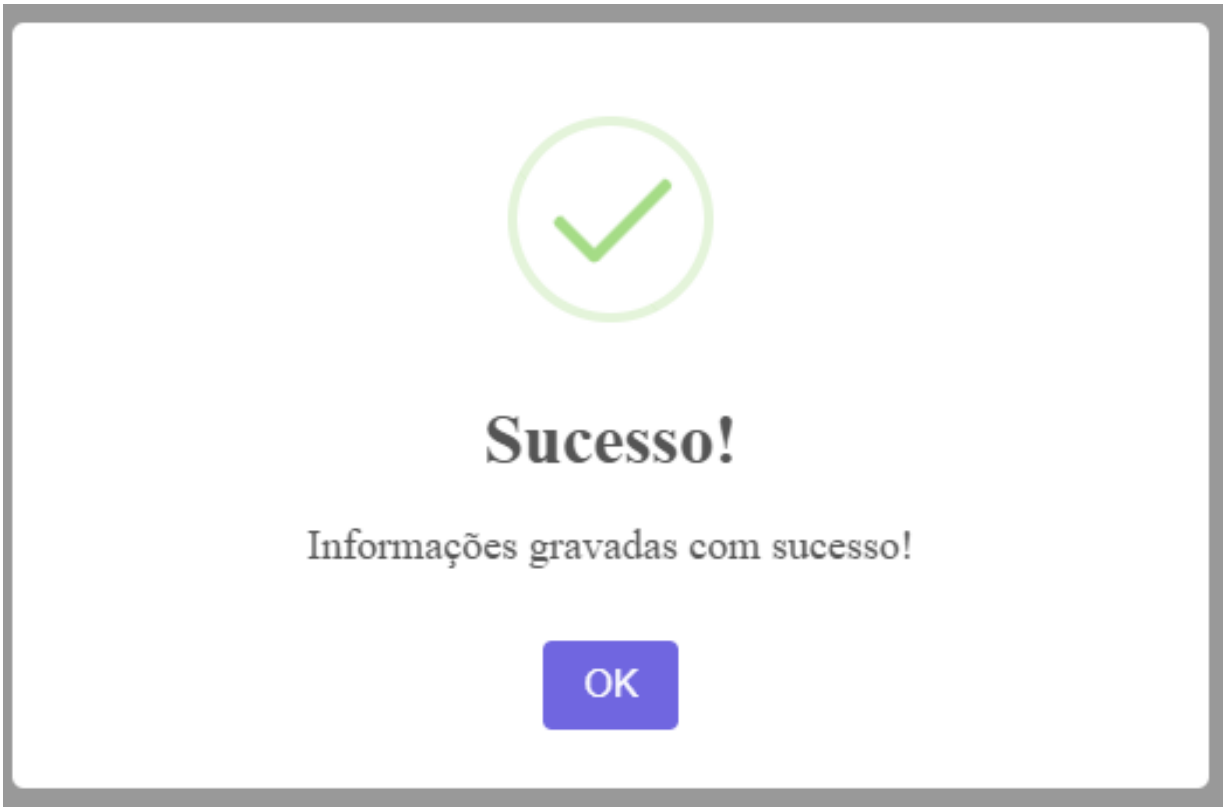


Figura 7.15: Exemplo do Sweet Alert

Bibliotecas são demais! Vamos conferir outra.

Toastify

Essa biblioteca permite exibir notificações discretas e personalizadas. Sua página inicial está no link https://apvarun.github.io/toastify-js e ela também possui uma excelente documentação. Essa biblioteca necessita de 2 links para sua instalação: um link de CSS e um de JS. Seu HTML precisará ter na tag `head` um link assim:

```
<link rel="stylesheet" type="text/css"
href="https://cdn.jsdelivr.net/npm/toastify-js/src/toastify.min.css">
```

E ao final do **body** haverá um link assim:

```
<script type="text/javascript" src="https://cdn.jsdelivr.net/npm/toastify-
js"></script>
```

Essa instalação foi um pouco diferente, mas em essência estamos linkando algumas coisas ao nosso projeto. Tudo isso sempre estará na documentação da biblioteca. Desta forma, estamos prontos para usar o *toastify*. Vamos usar um botão no HTML para acionar o toast. Então a função a seguir está sendo chamada por um botão em um HTML, ok?

```
function Notificacao(){
    Toastify({
        text: "Vamos para o site Espaço Dev?", //Texto do toast
        duration: 5000, //Duração em millisegundos
        destination: "https://espaco.dev", //Link ao clicar no toast
        newWindow: true, //Nova aba ao clicar no toast
        gravity: "top", // 'top' ou 'bottom' - Opções de posicionamento
vertical
        position: "right", // 'left', 'center' ou 'right' - Opções de
posicionamento horizontal
        stopOnFocus: true, // Se o mouse estiver em cima o toast não some
        style: {
            background: "blue", //Cor de fundo do toast
        },
        onClick: function(){} // Callback after click
    }).showToast();
}
```

O toast será algo como:

Figura 7.16: Exemplo visual do toastify

7.8 JavaScript tem frameworks?

Sim! O JavaScript tem vários frameworks que ajudam no desenvolvimento de aplicações web. Precisamos entender exatamente o que é um framework e como ele se diferencia de uma biblioteca.

O que é um framework?

Lá no início do livro, mencionamos brevemente o termo **framework** ao falar sobre ferramentas como Bootstrap e Chart.js. Agora, vamos retomar esse conceito com mais profundidade.

Um framework é um conjunto de códigos prontos que fornece uma estrutura base para desenvolver um tipo específico de aplicação. Ele define regras e padrões que os desenvolvedores seguem ao construir seus projetos, ajudando a organizar o código de forma padronizada e eficiente.

Se uma biblioteca é como uma caixa de ferramentas com funções que podemos usar sempre que precisamos, um framework é como um manual de montagem, que já nos dá o formato e a organização básica da aplicação.

Com um framework, não temos total liberdade para organizar o código como quisermos. Em vez disso, seguimos a estrutura que o ele impõe. Isso pode ser uma vantagem, pois padroniza o desenvolvimento, mas também pode ser uma limitação se precisarmos de mais flexibilidade.

Qual a diferença entre framework e biblioteca?

Característica	Biblioteca	Framework
Controle do fluxo	O programador chama funções da biblioteca quando precisa	O framework controla a estrutura e chama seu código
Flexibilidade	Você pode usá-la onde quiser	Você precisa seguir as regras do framework
Objetivo	Resolver problemas específicos (ex.: validação de dados, gráficos)	Fornecer uma base completa para o projeto
Exemplo no JS	SweetAlert, Toastify, Lodash, Axios	Angular, Vue.js, React

- Bibliotecas são usadas quando queremos algo pronto para facilitar o código.
- Frameworks são usados quando queremos um modelo estruturado para nosso projeto.

Frameworks populares no JavaScript

Agora que entendemos a diferença, vamos ver alguns frameworks famosos no mundo JavaScript:

- **Vue.js** – Criado para facilitar o desenvolvimento de interfaces interativas e reativas.
- **React** – Criado pelo Facebook, é muito usado para criar aplicações com componentes reutilizáveis.
- **Angular** – Criado pelo Google, é um framework completo para criar aplicações grandes e robustas.
- **Node.js** – Permite rodar JavaScript no servidor, sendo muito usado para back-end.

Cada um desses frameworks tem suas vantagens e desvantagens, mas todos têm o mesmo propósito: facilitar e padronizar o desenvolvimento de aplicações web

Agora você conhece bibliotecas e frameworks e já sabe diferenciá-las. Se precisar apenas de algumas funções extras, use uma biblioteca. Se quiser um modelo completo para desenvolver seu projeto, um framework pode ser a melhor escolha.

Futuramente, independente do que escolher, o importante é dominar o **JavaScript puro**, pois isso permitirá que você utilize qualquer biblioteca ou framework com facilidade. Nosso foco aqui é no JavaScript puro e espero que até aqui você já tenha se encantado com toda a força e robustez do JS puro.

Você é obrigado(a) a usar um framework? De forma alguma. Mas eles certamente oferecem facilidades. O objetivo deste livro é mostrar o poder do JavaScript e despertar em você a curiosidade para explorar ainda mais a área da tecnologia.

Se ao final você pensar algo como "Uau, o JavaScript já faz tanta coisa incrível... imagina o que posso construir com outras ferramentas!", então minha missão estará cumprida.

Parte III — Chegou a hora dos Dashboards!

Agora sim! Depois de toda a preparação, chegou o momento mais esperado deste livro: gerar Dashboards de verdade.

Você já aprendeu sobre o processo, conheceu as ferramentas e dominou o JavaScript. Nesta parte, vamos colocar tudo isso em prática para criar Dashboards completos e interativos. Vamos começar entendendo como tratar e organizar os dados para que fiquem prontos para visualização. Depois disso, partimos para a criação dos formatos — barras, linhas, mapas — sempre com exemplos práticos e fáceis de adaptar para seus próprios projetos.

Aqui você vai:

- Aprender a formatar os dados para o tipo certo de visualização.
- Integrar tudo que vimos em exemplos reais com HTML, CSS e JavaScript.
- Criar Dashboards de forma progressiva.
- Compor Dashboards bonitos e funcionais direto no navegador.

Se você seguiu até aqui, já tem tudo o que precisa. Agora é só dar vida aos dados e mostrar informações de forma clara, visual e impactante.

Vamos finalizar esse projeto com estilo!

Capítulo 8
Visualização e preparação dos dados

Uau! Foi uma grande caminhada até aqui. Nem começamos a parte mais divertida deste livro e, se você compreendeu todos os conceitos, saiba que fomos do mais básico ao mais complexo do

JavaScript puro. Passamos por tudo o que seria abordado em um curso de JavaScript!

Agora, você já é capaz de analisar qualquer código em JS e, mesmo que se depare com uma linguagem diferente, sua sólida base em lógica de programação facilitará muito o aprendizado.

Chegamos à etapa de transformar dados brutos em **informações**. Para isso, precisamos entender não apenas como gerar gráficos, mas como **preparar os dados** e pensar **visualmente**.

Vamos lá?

8.1 Fundamentos da visualização da informação

A visualização é uma ponte entre dados e o entendimento humano. Ela permite que padrões e tendências sejam percebidos com facilidade. Em vez de números em uma planilha, vemos histórias se formando.

Não se trata apenas de apresentar dados, mas sim de torná-los compreensíveis e eficazes para quem os interpreta. Para isso, aspectos como ambiente, percepção das cores e a forma como os elementos visuais são organizados desempenham um papel crucial.

O que quero dizer é que é totalmente possível **gerar emoções** por meio dos Dashboards. As seções a seguir demonstram que as escolhas de como gerar Dashboards não são apenas sobre a cor ou formato de que "eu mais gosto", mas sim que existe fundamento científico para apresentá-los melhor.

A percepção humana

Pense comigo: suponha que sua análise revelou que a empresa está enfrentando graves problemas financeiros e você decide exibir essas informações na cor verde. Isso faz sentido? Provavelmente, não. No contexto financeiro ocidental, verde representa crescimento e bons resultados. Mesmo que os números sejam preocupantes, se você apresentar os dados com essa cor, poderá diminuir o impacto real da informação.

O inverso também acontece. Imagine que você precisa suavizar um dado negativo para justificar um investimento. Talvez, neste caso, o verde seja a escolha ideal para tornar a informação mais aceitável.

Atenção! Não estamos falando em manipular informações por meio das cores ou formatos. Estamos falando sobre **comunicação eficiente** e de como um entendimento mais profundo desses aspectos pode ampliar nosso controle e reduzir a vulnerabilidade às mensagens que recebemos diariamente em jornais, redes sociais e televisão.

Segundo Colin Ware, professor do Departamento de Ciência da Computação da Universidade de *New Hampshire*, a percepção visual é orientada por fatores como contraste, agrupamento, forma e cor. Elementos bem posicionados reduzem a carga cognitiva e aceleram a compreensão da informação apresentada (WARE, 2021).

Ambiente e luminosidade

Fatores como iluminação do local e brilho da tela influenciam a leitura dos dados. Painéis em locais iluminados devem usar contrastes fortes; em ambientes escuros, uma paleta suave reduz o cansaço visual (WARE, 2021).

Cores e percepção

Cores podem comunicar significados, criar hierarquia e até evocar emoções. Porém, o uso inadequado pode causar confusão ou

transmitir mensagens erradas. A cor é poderosa, mas deve ser usada com responsabilidade (MUNZNER, 2014).

8.2 Agrupamentos, hierarquias e formas

A organização visual dos elementos é essencial. Agrupar dados similares, usar tamanhos proporcionais e manter uma hierarquia visual melhora a compreensão (por exemplo: Título > Destaque > Detalhes).

Narrativas e interação

Dashboards contam histórias. A disposição dos elementos, hierarquias visuais e interatividade são parte da narrativa. Um bom painel guia o leitor por uma jornada de descoberta, mesmo sem texto explicativo. A ordem, o destaque e a forma como os dados são revelados impactam essa experiência.

Um bom Dashboard conduz o usuário por uma narrativa. É possível, com poucos elementos, guiar o olhar, destacar fatos e provocar interpretações. Pense em um painel que mostra quedas de faturamento em vermelho e picos em verde. A história contada salta aos olhos (FEW, 2009).

8.3 Padronizando os dados

Antes de gerar gráficos, é essencial **padronizar os dados**. Isso significa organizá-los em um formato que facilite a leitura, a transformação e a visualização. Quase sempre será necessário converter formatos, limpar inconsistências e organizar os dados em arrays de objetos.

```
const dados = [
  { ano: 2023, estado: 'SP', imposto: 'IRPF', valor: 3500000 },
  { ano: 2023, estado: 'MG', imposto: 'IRPF', valor: 2950000 },
  ...
];
```

Quase sempre colocaremos os dados em arrays de objetos. Lembra que o objeto individual organiza as características de uma informação e que o array é uma lista de coisas? Quando juntamos os dois, temos uma lista de informações organizadas e isso é a organização perfeita para passarmos para qualquer Dashboard. É praticamente uma ordenação de informação tabelada, só que na programação.

Por que padronizar?

Nem sempre os dados que queremos apresentar estão prontos para uso. Às vezes, as fontes de informação apresentam inconsistências, dado monetário sem casa decimal, valor com separador de milhar em vírgula em vez de ponto, entre outras coisas que aterrorizam o(a) programador(a).

Não é incomum acontecer de demorarmos mais tempo organizando a informação do que efetivamente criando o Dashboard. Isso pode ser chato, mas acontece, e é melhor você saber disso antes e estar preparado.

Aliás, estou tão comprometido a preparar você para isso que vamos desenvolver, juntos, algumas funcionalidades em JavaScript que vão padronizar as informações e auxiliar na limpeza dos dados para uso.

Essa "limpeza" de dados geralmente tem como objetivo:

- Eliminar inconsistências;
- Facilitar a leitura de diferentes arquivos;
- Garantir reúso dos dados;
- Permitir centralização dos dados em arrays ou objetos reutilizáveis.

Como essas funcionalidades de limpeza e padronização serão usadas com frequência, recomendo que você as agrupe em um único arquivo JavaScript separado, como uma biblioteca de utilitários do seu projeto. Isso facilita a reutilização sempre que for necessário tratar dados novamente.

Uma boa analogia é pensar nas funções de limpeza de dados como ferramentas em uma obra. Um martelo, por exemplo, pode ser usado em diferentes momentos da construção e, por isso, fica guardado em uma caixa de ferramentas acessível. Da mesma forma, teremos um arquivo com funções reutilizáveis para padronizar os dados ao longo de todo o projeto.

No meu projeto aqui vou chamar este arquivo de `utilitarios.js`, mas fique à vontade para chamar seu arquivo com o nome que preferir.

Exemplo prático

Vamos supor que temos uma fonte de dados com esta estrutura:

```
Ano, Estado, Imposto, Valor
2023, SP, IRPF, 3500000
2023, RJ, IRPF, 1200000
2023, mg, Irpf, 12000.00
2023, PR, irpf, null
...
```

Observe que há dados com informações inconsistentes:

- algumas em maiúsculas, outras em minúsculas;
- com casa decimal e outras sem;
- `null` em vez de um valor.

É muito estranho haver um `null`. Como isso pode ser um dado válido? Realmente não é, mas temos que estar prontos caso aconteça. Talvez aquele dado específico ainda não tenha sido

apurado ou não foi lançado e ele simplesmente não existe. Um dado desse no meio da criação do Dashboard pode quebrá-lo inteiro.

Aqui estamos observando um exemplo com apenas quatro linhas de informação, mas vale lembrar que podem ser milhares de registros — e nem sempre será fácil identificar um `null` entre os valores. Esse é mais um motivo para iniciarmos nossa ferramenta de limpeza com funções úteis adaptadas a este cenário.

Podemos corrigir valores monetários. A função a seguir formata qualquer valor em um padrão aceitável para nossa programação. Se o valor vier com ponto ou vírgula, a função o corrige para um formato numérico adequado. Inclusive, se o número não existir (se vier `null`), ele retorna algo minimamente válido, que é o zero.

```
// Ex: "1.234,56" => 1234.56
function normalizarValorMonetario(valor) {
  if (!valor) return 0;
  return parseFloat(
    valor.toString()
      .replace(/\./g, '') // remove separador de milhar
      .replace(',', '.')  // troca vírgula por ponto decimal
  );
}
```

Também devemos padronizar maiúsculo ou minúsculo. Não seria legal algo que mostra o estado "SP" em um ponto, depois mostra estado "mg", depois o estado "Pr". O mesmo vale para o exemplo do imposto: em um momento, exibe "IRPF", depois exibe "Irpf" etc.

```
//Ex: "sp" => "SP" ou "Sp" => "SP"
function padronizarMaiusculo(texto) {
  return texto ? texto.toString().toUpperCase() : '';
}

//Não usado neste exemplo, mas é bom termos uma função para minúsculo
function padronizarMinusculo(texto) {
```

```
  return texto ? texto.toString().toLowerCase() : '';
}
```

Para usarmos qualquer uma dessas ferramentas, vamos ter que passar todos os dados por essas funções. Então, vamos imaginar que toda a programação está acontecendo e chegou o momento de guardar o valor do imposto em uma variável `valor`. Em vez de você simplesmente fazer assim:

```
const valor = "1.234,56";
```

Você faria algo assim:

```
const valor = normalizarValorMonetario("1.234,56");
```

Com isso, você passou o valor pela sua "ferramenta de limpeza" e a variável `valor` estará formatada. Depois de passarmos todas as informações da tabela pelas nossas ferramentas utilitárias, os dados que anteriormente estavam assim:

```
Ano, Estado, Imposto, Valor
2023, SP, IRPF, 3500000
2023, RJ, IRPF, 1200000
2023, mg, Irpf, 12000.00
2023, PR, irpf, null
```

Estarão agora padronizados:

```
Ano, Estado, Imposto, Valor
2023, SP, IRPF, 3500000
2023, RJ, IRPF, 1200000
2023, MG, IRPF, 12000.00
2023, PR, IRPF, 0.00
```

E, na sequência, só falta colocarmos em objetos para elas ficarem assim:

```
const dados = [
  { ano: 2023, estado: 'SP', imposto: 'IRPF', valor: 35000.00 },
  { ano: 2023, estado: 'RJ', imposto: 'IRPF', valor: 12000.00 },
  { ano: 2023, estado: 'MG', imposto: 'IRPF', valor: 12000.00 },
```

```
  { ano: 2023, estado: 'PR', imposto: 'IRPF', valor: 0.00 },
];
```

Para isso, é só aplicar as funções de forma sistemática sobre os dados que foram lidos, geralmente linha a linha, criando um novo objeto já limpo e pronto para uso.

```
const dadosBrutos = [
  {ano: '2023', estado: 'SP', imposto: 'IRPF', valor: '3500000'},
  {ano: '2023', estado: 'RJ', imposto: 'IRPF', valor: '1200000'},
  {ano: '2023', estado: 'mg', imposto: 'Irpf', valor: '12000.00'},
  {ano: '2023', estado: 'PR', imposto: 'irpf', valor: null }
];

const dados = dadosBrutos.map(linha => ({
  ano: linha[0],
  estado: padronizarMaiusculo(linha[1]),
  imposto: padronizarMaiusculo(linha[2]),
  valor: normalizarValorMonetario(linha[3])
}));
```

Pronto! A partir disso, podemos criar filtros, mapear os dados, agrupar valores e alimentar qualquer tipo de Dashboard.

Boas práticas

Também vale mencionar, de forma breve, que é uma boa prática padronizar a nomenclatura de variáveis, tipos e até mesmo o escopo.

- Use nomes de campos padronizados (`ano` , `valor` , `categoria` etc.);
- Sempre converta strings numéricas para `Number()` ou `parseFloat()` para que o seu Dashboard não confunda posicionamento numérico com texto;
- Armazene os dados tratados em variáveis ou listas globais, como `dadosPadronizados` , por exemplo. Isso permite usar os

dados tratados mais de uma vez, economizando processamento.

Evite, por exemplo, nomes como `val`, `dado1`, `xpto` ou `temporario`, que não explicam o que está sendo armazenado. Isso dificulta a leitura e o entendimento, especialmente quando o código cresce.

Vamos fazer isso também nas gerações de Dashboards do próximo capítulo.

8.4 Escolhendo os tipos de Dashboards

Agora que os dados estão organizados e padronizados, precisamos escolher **como mostrá-los**. Não é uma obrigatoriedade, mas alguns tipos já possuem uma finalidade bem definida.

Tipos comuns e seus usos

Tipo de Dashboard	Melhor uso
Barras	Comparar categorias
Linhas	Evolução temporal
Pizza	Proporções simples
Área	Acúmulo ao longo do tempo
Radar	Comparações entre variáveis

Sua escolha deve considerar aspectos como a clareza visual do gráfico, a legibilidade dos rótulos e a simplicidade da forma. Por exemplo, evite gráficos de pizza com muitas fatias. Prefira cores contrastantes e organize os elementos de maneira intuitiva.

8.5 Checklist para bons Dashboards

Antes de gerar seus Dashboards, lembre-se dessas boas práticas:

- Uso consciente de cores;
- Hierarquia visual clara;
- Títulos e rótulos explicativos;
- Tipo de gráfico adequado aos dados;
- Organização limpa e intencional.

Este pequeno checklist já resolve 80% dos erros cometidos em visualizações.

Dashboards eficazes equilibram riqueza informacional com simplicidade visual. Conforme diz Edward Tufte, "a excelência gráfica consiste em transmitir o máximo de ideias no menor espaço possível sem confundir o leitor". Pense nisso: ao adicionar muitos elementos em um gráfico, eles precisam servir à clareza, não apenas ao volume.

Neste capítulo, vimos como os fundamentos da visualização da informação — baseados em autores como Colin Ware — são essenciais para dar sentido à criação de dashboards. Não se trata apenas de exibir dados: é possível contar histórias, transmitir emoções e facilitar a compreensão por meio de escolhas visuais conscientes.

> Para quem deseja se aprofundar mais, a própria Casa do Código possui livros excelentes sobre *Data Visualization*, com abordagens complementares a esta introdução.

No próximo capítulo, começaremos a colocar tudo isso em prática com a geração efetiva dos nossos Dashboards!

8.6 Referências do capítulo

FEW, Stephen. *Now You See It: Simple Visualization Techniques for Quantitative Analysis*. 2009.

MUNZNER, Tamara. *Visualization Analysis and Design*. CRC Press, 2014.

TUFTE, Edward R. *The Visual Display of Quantitative Information*. Graphics Press, 1983.

WARE, Colin. *Information Visualization: Perception for Design*. 4. ed. Morgan Kaufmann, 2021.

Capítulo 9
Gerando os Dashboards

Chegou o momento mais aguardado: colocar a mão na massa e transformar os dados padronizados em Dashboards Interativos! Os Dashboards são o produto final de qualquer análise de dados ou software de visualização. Para que eles existam, é necessário que haja dados predecessores. Eles são a "cereja do bolo" de todo aquele amontoado de informações que, até agora, estava sendo organizado, estruturado e preparado.

Se você acompanhou os capítulos anteriores, seu ambiente já está pronto para essa nova fase. A partir daqui, veremos diferentes formas de criar Dashboards e, como você já sabe, todos eles têm algo em comum: **dados, muitos dados**.

Vamos criar diversos modelos — financeiros, tributários, percentuais e outros — para que você veja, na prática, como adaptar os formatos aos mais diversos contextos. Assim, você poderá analisar qual abordagem se encaixa melhor nos seus próprios dados e aplicar da forma que preferir. Seja usando exatamente os exemplos deste livro ou combinando as ideias com sua própria criatividade, você terá em mãos todas as ferramentas para construir Dashboards incríveis!

Agora vamos integrar a biblioteca `Chart.js` ao nosso projeto e começar.

Em todos os arquivos com extensão `.html` adicionaremos as seguintes linhas ao final do arquivo, antes de fechar a tag `</body>`:

```
<script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
<script src="utilitarios.js"></script>
```

A primeira tag adiciona ao nosso projeto a biblioteca com componentes visuais pré-processados e prontos para uso. A segunda importa as ferramentas genéricas de limpeza que criamos no

capítulo anterior — lembrando que, no meu exemplo, dei o nome de `utilitarios.js`, mas você pode ter usado outro nome.

9.1 Como acompanhar os códigos deste capítulo

À medida que gerarmos os Dashboards, é possível que os códigos fiquem extensos e com vários arquivos. Para melhorar a experiência deste livro, criei um repositório somente com o que gerarmos aqui. Assim, você pode ler as informações e testar os códigos aqui expostos.

Tudo está disponível no repositório GitHub:

Repositório oficial do livro:
https://github.com/luctauil/livro_dashboards_com_js.git

Figura 9.1: QR Code repositório - Dashboards

Para facilitar a criação dos Dashboards, adotaremos uma abordagem modular: um Dashboard por página HTML. Cada página terá um esqueleto semelhante, o que reforça a separação entre estrutura (HTML) e dinamismo (JavaScript).

Dessa forma, construiremos um modelo base de HTML que será reaproveitado em todos os Dashboards. A estrutura será sempre a mesma — o que muda de um Dashboard para outro será o arquivo JS com os dados e a lógica visual específica.

Nosso HTML base será:

```
<div class="container py-5">
  <h1 class="text-center mb-4">Dashboard 1</h1>
  <p class="lead text-center">Visualização interativa de dados</p>

  <div class="my-5">
      <canvas id="meu_dashboard"></canvas> <!-- Geração nesta tag -->
```

```
  </div>

</div>

<!-- Chart.js -->
<script src="https://cdn.jsdelivr.net/npm/chart.js"></script>

<!-- Nossas funções de limpeza -->
<script src="scripts/utilitarios.js"></script>

<!-- Script do Dashboard desta página -->
<script src="scripts/01_dashboard.js"></script>
```

> O código descrito é o conteúdo de dentro do `<body>`. Lembre-se de incluir a estrutura completa do HTML (`<!DOCTYPE html>`, `<html>`, `<head>`, `<title>` etc.) ao redor desse conteúdo.

A partir deste ponto no livro, esse será o esqueleto base para todos os Dashboards desenvolvidos. Cada Dashboard terá sua própria página HTML e um script JavaScript correspondente.

O conteúdo será renderizado dentro da tag `<canvas>`, e você pode adaptar livremente o restante da estrutura HTML conforme as necessidades do seu projeto.

Por exemplo: é perfeitamente possível ter mais de um Dashboard na mesma página, bastando adicionar outras tags `<canvas>` com IDs diferentes e scripts que apontem corretamente para cada um deles. Nosso foco aqui será em exemplos simples e claros, mantendo a estrutura enxuta para facilitar a compreensão do que está sendo desenvolvido.

O que é importante lembrar:

- A tag `<canvas>` é o espaço onde o Dashboard será renderizado;
- As três inclusões via `<script>` seguem a ordem:

- A biblioteca Chart.js permite gerar desenhos gráficos na página web;
- O arquivo `utilitarios.js`, com funções genéricas de limpeza e padronização dos dados;
- O arquivo específico deste Dashboard, como `01_dashboard.js`.

Vale mencionar que toda a estrutura de exibição que utilizaremos aqui é baseada na biblioteca **CHART.JS** (https://www.chartjs.org/). Essa biblioteca é extremamente poderosa, gratuita, leve e amplamente utilizada em aplicações reais.

Se quiser se aprofundar mais ou explorar tipos de Dashboards que não cobrimos aqui, recomendo uma visita à documentação oficial do Chart.js, em https://www.chartjs.org/docs/latest/.

9.2 Seu primeiro Dashboard

Vamos gerar um primeiro Dashboard simples considerando que gostaríamos de visualizar a somatória de valores de algumas regiões. Vamos utilizar o formato de barras, um comparador útil, considerando que os valores já estão em um array no nosso código.

```
const categorias = ['SP', 'RJ', 'MG', 'PR']; //Categorias - rótulos do eixo X
const valores = [70000, 120352.99, 87000, 62998.87]; //Valores - dados do eixo Y
const cores = ['blue', 'green', 'orange', 'red']; //Cores - tonalidades para cada barra

const ctx = document.getElementById('meu_dashboard').getContext('2d');
//Captura do elemento canvas que está no nosso HTML

/* A seguir, toda a criação do primeiro Dashboard. Esse formato é
```

```
padronizado para o Chart.js e só temos que associar nossas variáveis aos
eixos de exibição */
const meu_dashboard = new Chart(ctx, {
    type: 'bar', //Define o tipo de Barras
    data: {
        labels: categorias, //Rótulos do eixo X
        datasets: [{
            label: 'Valores', //Rótulo do eixo Y
            data: valores, //Dados do eixo Y
            backgroundColor: cores, //Cores para cada barra
        }]
    },
    options: { //Opções adicionais que podem ser configuradas
        plugins: {
            legend: {
                display: false // Habilita ou desabilita legenda padrão
            }
        }
    }
});
```

Este código vai gerar o seguinte Dashboard:

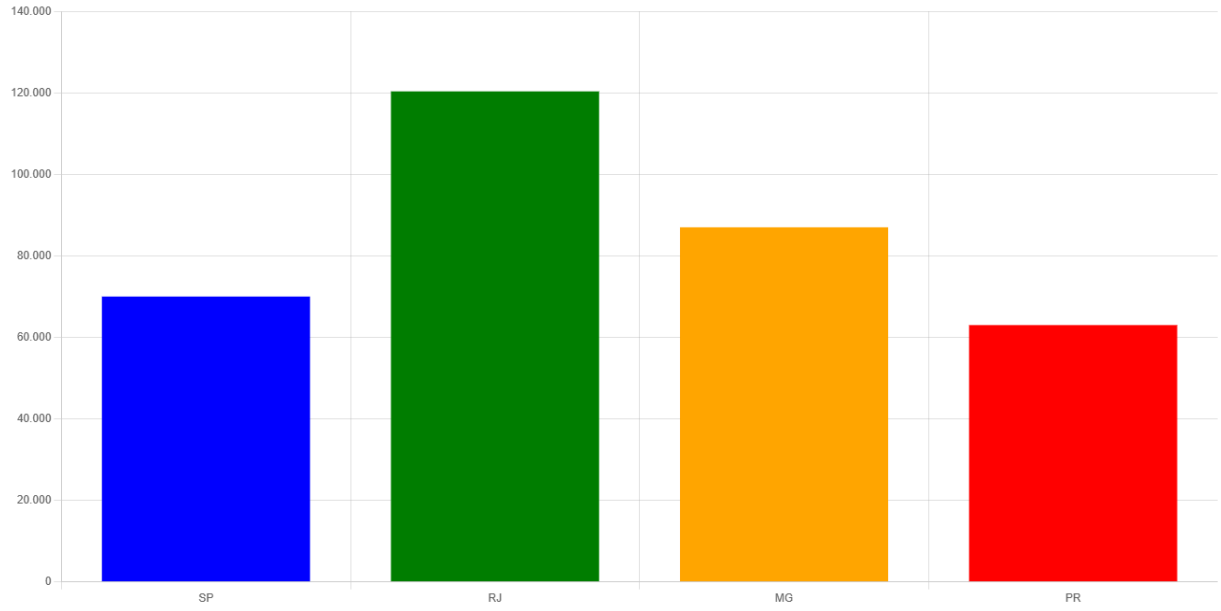


Figura 9.2: Dashboard de Barras

Observe que na elaboração deste Dashboard temos uma construção que é feita automaticamente. O Chart.js nos oferece uma estrutura pronta. Nossa responsabilidade aqui é apenas associar corretamente os dados que queremos visualizar: os rótulos (`categorias`), os valores e as cores.

Os primeiros valores de cada array se relacionam, assim como o segundo, terceiro e quarto. Se houver um quinto estado, basta adicionar ao final de cada array a informação relacionada. Vamos supor que você altere os arrays para:

```
const categorias = ['SP', 'RJ', 'MG', 'PR', 'RS', 'BA'];
const valores = [70000, 120352.99, 87000, 62998.87, 50000, 23000];
const cores = ['blue', 'green', 'orange', 'red', 'purple', 'pink'];
```

Com isso, seu Dashboard passaria a exibir mais estados, ampliando a análise com os novos dados inseridos.

Experimente alterar a ordem dos estados, adicionar um novo ou deixar um valor em branco para ver o que acontece. Essas pequenas variações ajudam a consolidar o entendimento sobre a ligação entre os arrays.

9.3 Um Dashboard dinâmico

Faremos o próximo Dashboard de forma um pouco mais dinâmica, supondo que os dados não estejam predefinidos nos arrays ou que você queira montá-los e testá-los de formas diferentes. Vamos adicionar alguns campos e dinamizar um pouco mais? Para esse exemplo, vamos simular um Dashboard de total de vendas feitas por vendedor.

Olhe este Dashboard em formato de pizza, criado dinamicamente. Cada fatia representa um vendedor e o usuário consegue ir

adicionando os vendedores e seus valores de venda (e até mesmo selecionar as cores).

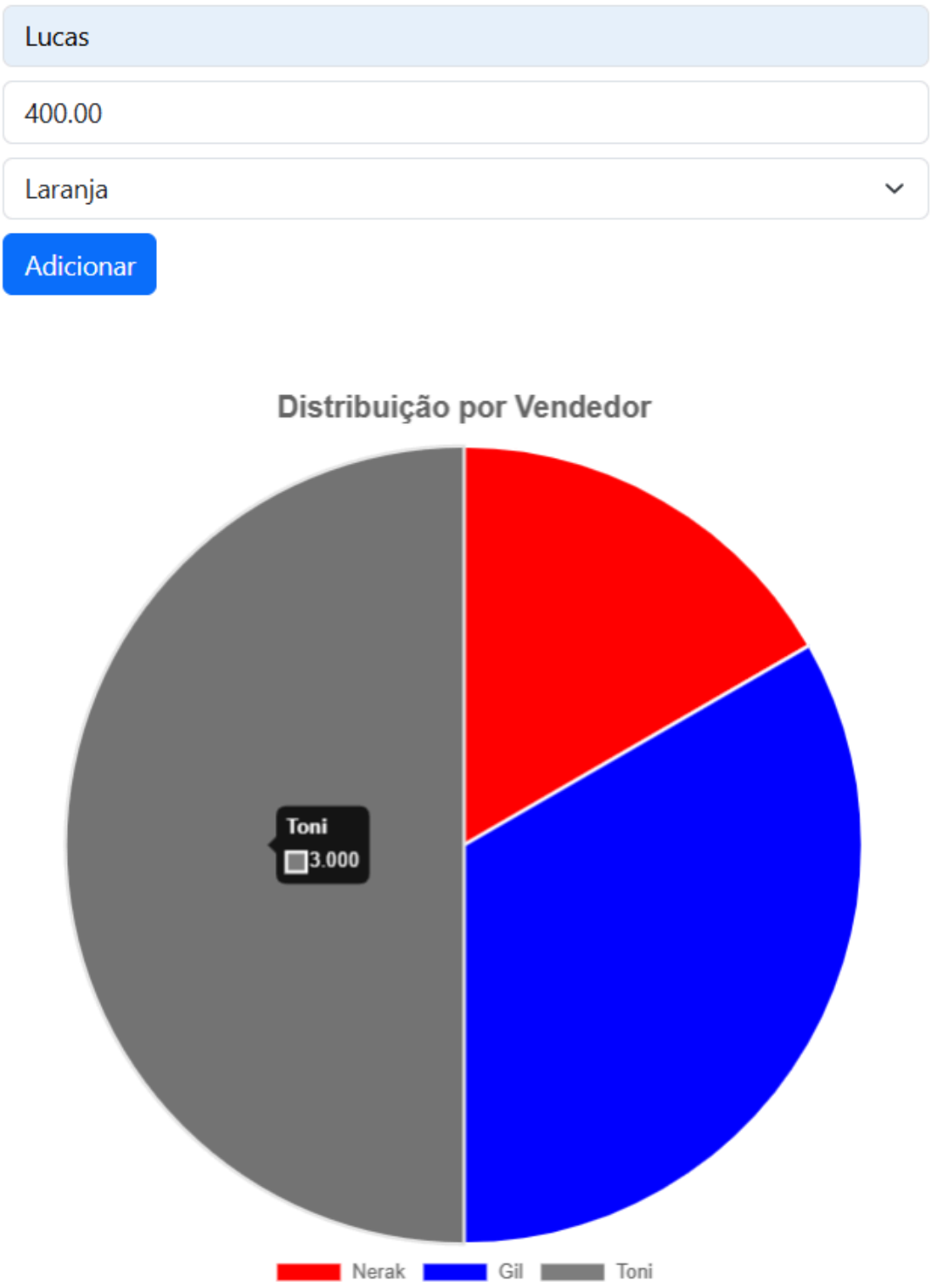


Figura 9.3: Dashboard de Pizza

As legendas são automaticamente geradas e interativas. Você consegue clicar e "desclicar" nos vendedores dando dinamismo.

O princípio é o mesmo do que fizemos anteriormente. Poucas coisas mudam. Vamos analisar: no HTML, foi preciso adicionar os campos para o usuário digitar o nome do vendedor, o valor da venda, a cor associada a ele e um botão para adicionar aquela informação. Então, no HTML, foi necessário adicionar os campos:

```
<div>
    <input type="text" id="vendedor" class="form-control"
placeholder="Digite o nome do vendedor...">
    <input type="text" id="valor" class="form-control mt-2"
placeholder="Digite o valor total da venda...">

    <select id="escolha_cor" class="form-select mt-2">
        <option value="red">Vermelho</option>
        <option value="blue">Azul</option>
        <option value="green">Verde</option>
        <!-- ... Só ir repetindo conforme as opções que deseja -->
    </select>

    <!-- Um botão chamando uma ação que adiciona os valores ao Dashboard
de pizza -->
    <button id="adicionar" class="btn btn-primary mt-2"
onclick="AdicionarValor()">Adicionar</button>
</div>
```

E o JS tem algumas novidades devido ao uso dos campos adicionados, mas a estrutura da geração da pizza é muito similar à das barras:

```
//Os arrays de rótulos começam vazios, pois serão adicionados pelo usuário
const vendedores = [];
const valores = [];
const cores = [];

// Captura do elemento <canvas> no HTML onde o Dashboard será renderizado
const ctx = document.getElementById('meu_dashboard').getContext('2d');
```

```
/* Criação do Dashboard tipo 'pizza' (pie). Este formato distribui os
valores proporcionalmente em um círculo. */
const meu_dashboard = new Chart(ctx, {
    type: 'pie', // Define tipo como pizza
    data: {
        labels: vendedores, // Rótulos das fatias
        datasets: [{
            data: valores, // Valores de cada fatia
            backgroundColor: cores, // Cores das fatias
        }]
    },
    options: {
        plugins: {
            legend: {
                display: true, // Exibe a legenda com cores e rótulos
                position: 'bottom' // Posição da legenda
            },
            title: {
                display: true,
                text: 'Distribuição por Vendedor',
                font: {
                    size: 18
                }
            }
        }
    }
});

//A função que é chamada pelo botão
function AdicionarValor(){
  let vendedor = document.getElementById('vendedor').value; // Captura o
nome do vendedor
  let valor = parseFloat(document.getElementById('valor').value); //
Captura o valor a ser adicionado
  let cor = document.getElementById('escolha_cor').value; // Captura a cor
da fatia

  vendedores.push(vendedor); // Adiciona o vendedor ao array
```

```
  valores.push(valor); // Adiciona o valor ao array
  cores.push(cor); // Adiciona a cor ao array

  meu_dashboard.update(); // Atualiza o Dashboard com os novos dados
  document.getElementById('vendedor').value = ''; // Limpa o campo do
vendedor
  document.getElementById('valor').value = ''; // Limpa o campo do valor
}
```

Muito legal, não é?! Colocamos alguns campos a mais no HTML, uma função para um botão e alguns parâmetros de valores na construção do Dashboard. Talvez você pense que a alteração foi grande e que o código aumentou cerca de 20 linhas, mas acredite: para todo o dinamismo que criamos, foram poucas mudanças no universo da programação.

9.4 E se tivermos outras fontes de dados?

Nossos dois primeiros Dashboards foram legais, mas ambos usaram poucos dados ou até mesmo dados digitados. Quero demonstrar Dashboards com várias fontes de informações, incluindo informações online de APIs. Vamos ver um formato desses? É muito legal!

O próximo Dashboard é para fãs de criptomoedas e, mesmo que não seja seu caso, o exemplo é muito bom, pois a ideia aqui é obter informações atuais de uma API e mostrar se o preço aumentou ou diminuiu. Quando digo "obter o preço atual", me refiro a fazer o JavaScript buscar as informações daquele momento em que a pessoa acessou o Dashboard.

Vamos utilizar uma parte **gratuita** da API do Mercado Bitcoin — uma corretora brasileira de compra e venda de criptomoedas. O trecho que usaremos está disponível por meio de um link público:

```
https://api.mercadobitcoin.net/api/v4/tickers?symbols=BTC-BRL
```

Se você copiar esse link e colar diretamente no seu navegador você verá, a princípio, algumas informações confusas. Mas, ao olhar com calma, reconhecerá esse padrão que estudamos no capítulo 4. Você verá algo parecido com isso:


```
[{"pair":"BTC-
BRL","high":"517247.00000000","low":"496572.00000000","vol":"27.88543250",
"last":"507439.00000000","buy":"507561.00000000","sell":"507823.00000000",
"open":"498064.00000000","date":1745273187}]
```


É um link que não exibe uma página. Ele traz informações no formato JSON e é isso que geralmente as APIs fazem. Vamos entender o que é cada informação desta API:

- `"pair": "BTC-BRL"` — é o par de conversão. Isso indica que estamos olhando a cotação do Bitcoin (BTC) em relação ao Real Brasileiro (BRL). É o ativo que pedimos na URL (no link).
- `"high": "517247.00"` — este foi o maior valor atingido pelo Bitcoin no período analisado (geralmente nas últimas 24h).
- `"low": "496572.00"` — este foi o menor valor registrado no mesmo período. Assim, conseguimos ter uma ideia de variação de preço.
- `"vol": "27.88543250"` — esse é o volume total negociado de Bitcoin no período. Ou seja, aproximadamente 27,88 BTC foram comprados e vendidos nas últimas 24 horas.
- `"last": "507439.00"` — este é o último preço registrado da moeda. Em geral, é esse valor que você vê como "cotação atual".
- `"buy": "507561.00"` — valor de compra atual. Ou seja, o maior preço que alguém está disposto a pagar no momento.
- `"sell": "507823.00"` — valor de venda atual. O menor preço que alguém está disposto a vender (aqui temos o famoso "spread", a diferença entre os preços de compra e venda).
- `"open": "498064.00"` — esse foi o valor de abertura, ou seja, o preço do Bitcoin no começo do período considerado.
- `"date": 1745273187` — essa é a data e hora da cotação, mas está em formato de *timestamp Unix* (segundos desde 01/01/1970).

Vamos precisar converter essa data e vamos usar o JS pra isso.

> O formato timestamp Unix representa a quantidade de segundos desde 1º de janeiro de 1970 (UTC). É uma forma padronizada e numérica de registrar datas e horários em sistemas. Para torná-lo legível, usamos linguagens como JavaScript para converter em uma data comum.

Vimos todos os campos que a API trouxe, mas nem sempre precisamos de todas as informações. Nosso interesse é apenas no valor atual. Portanto, dentre todas essas informações da API, usaremos somente o campo `last`, que representa a cotação atual.

Vamos ver o Dashboard pronto e seguimos para o código:

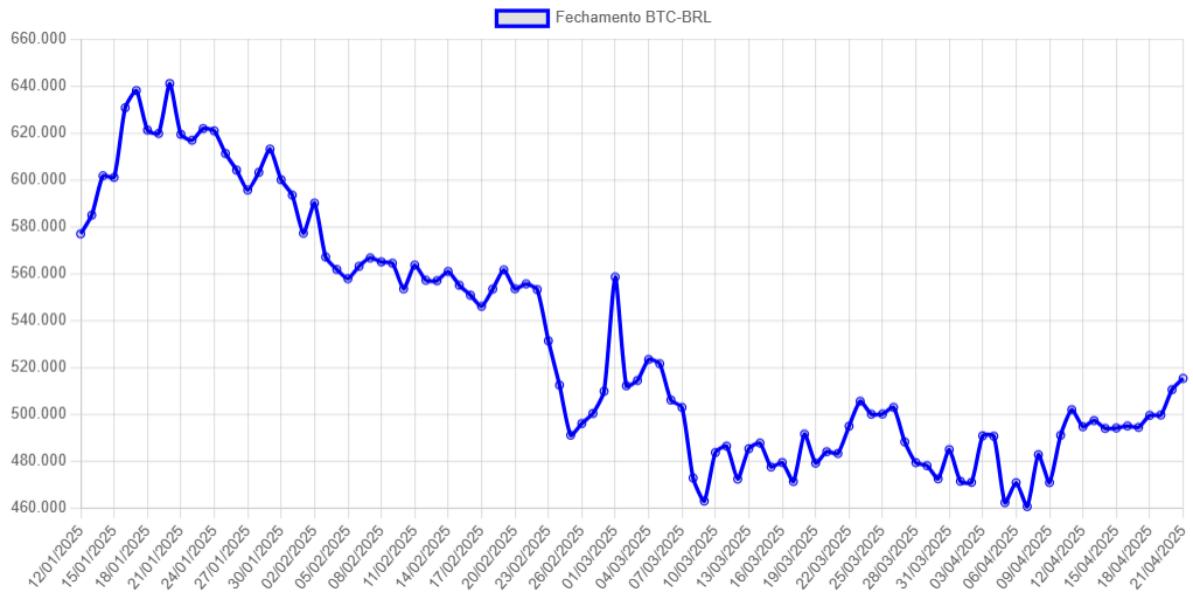


Figura 9.4: Dashboard de linhas

É um exemplo perfeito de qualquer análise financeira. A relação do preço de alguma coisa (eixo vertical) em um espaço temporal (eixo horizontal).

Nesta imagem, estamos vendo um intervalo de aproximadamente 100 dias. Mas o código do Dashboard nos dá alguns parâmetros interessantes. Vamos dividir em partes:

- As variáveis que vamos usar:

```
const dias = 100; //Uma variável fácil de alterar se quisermos reduzir ou aumentar o intervalo
const baseURL = 'https://api.mercadobitcoin.net/api/v4/candles'; //O link da API que fornece os dados
const historico = []; //O array vazio que vai receber os dados para posterior montagem do Dashboard

buscarUltimosDias(dias); //A chamada da função de busca com a quantidade de dias que será buscada.
```

- A busca e preparação dos dados (observe que entre códigos deixei a explicação das linhas nos comentários):

```
async function buscarUltimosDias(dias) {
    const hoje = new Date(); //Cria um objeto de data com a data de agora
    hoje.setDate(hoje.getDate() - 1); //Reduz 1 dia da data atual
    hoje.setHours(23, 59, 59, 999); //Define a hora para o final do dia (23:59:59.999)

    const timestampTo = Math.floor(hoje.getTime() / 1000); //Converte a data para timestamp em segundos
    const timestampFrom = timestampTo - ((dias) * 86400); //Calcula o timestamp de dias atrás (86400 segundos em um dia)

    /* Essas conversões de datas temos que fazer, pois é o formato que a API obriga. Quando estamos usando uma API externa, temos que programar conforme as regras da API */

    const url = `${baseURL}?symbol=BTC-BRL&resolution=1d&from=${timestampFrom}&to=${timestampTo}`; //link da API

    const resposta = await fetch(url); //JavaScript efetuando a busca dos dados na API
    const json = await resposta.json(); //Resposta guardada em uma
```

```
variável

    /*Os dados estão todos na variável json. O for "pega" esses dados e
formata dentro de um array de objetos. É a preparação dos dados. */

    for (let i = 0; i < dias; i++) { //A organização dos dados acontece na
quantidade de dias que pedimos.
        /* O trecho a seguir vai montar um array de objetos com o seguinte
formato:
        [ ...
          {data: 'XX/XX/XXXX', fechamento: XXX.XX},
          {data: 'XX/XX/XXXX', fechamento: XXX.XX},
          {data: 'XX/XX/XXXX', fechamento: XXX.XX},
        ... ]
        */

        historico.push({
            data: new Date(json.t[i] * 1000).toLocaleDateString('pt-BR'),
            fechamento: parseFloat(json.c[i]), // valor representativo do
dia
        });

        /* json.t[i] é o formato que a API retorna. Temos que respeitar o
nome da variável da API.
        o formato é multiplicado por 1000 para converter de segundos para
milissegundos, e depois usamos toLocaleDateString para formatar a data.
        json.c[i] é o valor de fechamento do dia. O valor é convertido
para float para garantir que seja um número decimal. */
    }
    MontarDashboard(); //Com os dados prontos no array historico, podemos
chamar a montagem do Dashboard
}
```

- A montagem do Dashboard na tela:

```
function MontarDashboard() {

// Extrai os dados do array original para arrays separados por eixo do
Dashboard
const labels = historico.map(d => d.data); // ['18/04', '19/04', ...,
```

```
'Hoje']
const valores = historico.map(d => d.fechamento); // [502000, 508000, ...,
520350]

// Cria o Dashboard
const ctx = document.getElementById('meu_dashboard').getContext('2d');
const dashboard = new Chart(ctx, {
    type: 'line', //Só muda para o tipo linha.
    data: {
        labels: labels,
        datasets: [{
            label: 'Fechamento BTC-BRL',
            data: valores,
            fill: false,
            borderColor: 'blue',
        }]
    },
});

}
```

Agora, com o código pronto, você consegue alterar alguns parâmetros e fazer sua própria análise. Você pode mudar a quantidade de dias de análise só alterando a variável `dias` e o Dashboard já terá outra aparência. Inclusive, você pode colocar um `input` na tela para que o usuário digite a quantidade de dias que ele deseja analisar.

Outro ponto muito legal deste Dashboard é que ele está sempre atualizado. Como usamos o objeto `Date()` do JavaScript, a consulta à API sempre será pela data atual. Você não tem um trabalho secundário de atualizar as informações.

9.5 Comparar mais dados

Ainda neste painel, vamos comparar mais valores. Na função de criação do Dashboard, observe este trecho padrão:

```
datasets: [{
    label: 'Fechamento BTC-BRL',
    data: valores,
    fill: false,
    borderColor: 'blue',
}]
```

Note que este trecho `datasets` existe em todos os Dashboards e que, embora as características sejam fixas (`label` , `data` , `fill` etc.), ele não passa de um array de objetos. Ou seja, se colocarmos mais objetos dentro desse array seguindo os nomes, eles vão ser renderizados simultaneamente no mesmo Dashboard.

Então, vamos fazer algumas adaptações no código para adicionar mais moedas neste mesmo Dashboard. Para que isso faça sentido, a relação temporal (eixo horizontal) será a mesma. Será um comparativo de diversas moedas no mesmo intervalo de tempo.

No trecho de variáveis, vamos criar um array de moedas. Antes, buscávamos apenas o Bitcoin; agora, vamos buscar várias moedas.

```
const moedas = ['XRP-BRL', 'DOGE-BRL', 'ADA-BRL', 'USDT-BRL', 'AXL-BRL',
'CAKE-BRL', 'UMA-BRL'];
const historicoPorMoeda = [];
```

Por enquanto, a única alteração foi na nomenclatura da variável, para torná-la mais coerente com o que estamos fazendo. Além disso, criamos um novo array onde definimos as criptomoedas que queremos buscar na API.

A escolha dessas moedas se deve ao fato de que, no momento em que escrevo este capítulo, elas apresentam valores relativamente próximos entre si, o que torna a comparação mais interessante do ponto de vista visual e dinâmico.

Você vai notar que removi o Bitcoin da lista justamente por ele destoar muito em valor das demais criptos — o que poderia prejudicar a proporção entre elas no gráfico. Mas sinta-se à vontade para substituir ou incluir as moedas que achar mais relevantes.

As criptos serão:

- XRP-BRL - Ripple;
- DOGE-BRL - Dogecoin;
- ADA-BRL - Cardano;
- USDT-BRL - Tether;
- AXL-BRL - Axelar;
- CAKE-BRL - PancakeSwap;
- UMA-BRL - UMA.

Também precisamos fazer algumas alterações na função que realiza a busca na API. As configurações de tempo permanecem as mesmas, mas agora precisamos fazer uma busca separada para cada criptomoeda daquela mesma relação temporal.

Na prática, é a mesma lógica da função anterior — a diferença é que agora repetiremos a busca para cada moeda. E é aí que o for entra: ele é ideal para esse tipo de repetição. Lembra que o `for` executa um bloco de código várias vezes?

Portanto, dentro da função `buscarUltimosDias`, vamos ajustar o trecho responsável pela busca para que ele percorra o array de moedas e execute uma requisição para cada uma delas.

```
for (const moeda of moedas) { //Fará uma busca para cada moeda.

  //O link da API tem uma adaptação para que busque cada moeda
  const url = `${baseURL}?
symbol=${moeda}&resolution=1d&from=${timestampFrom}&to=${timestampTo}`;
  const resposta = await fetch(url);
  const json = await resposta.json();

  historicoPorMoeda[moeda] = []; //O array que guardava os valores de uma
```

```
criptomoeda agora se torna um array de arrays e guarda os valores por
moeda.

  for (let i = 0; i < dias; i++) {
      historicoPorMoeda[moeda].push({
          data: new Date(json.t[i] * 1000).toLocaleDateString('pt-BR'),
          fechamento: parseFloat(json.c[i])
      });
  }
}
MontarDashboard(); //Dados prontos. Só chamar a montagem do Dashboard.
```

Na função que renderiza o Dashboard, precisamos fazer algumas adaptações para que sejam enviados mais `datasets`.

```
function MontarDashboard() {
    const ctx = document.getElementById('meu_dashboard').getContext('2d');

    //labels pega as datas apenas da primeira criptomoeda, pois sabemos
que o período de todas é igual.
    const labels = historicoPorMoeda[moedas[0]].map(d => d.data);

    const cores = ['blue', 'green', 'orange', 'gray', 'purple', 'red',
'black']; // uma cor por moeda.

    //Montagem de um dataset por criptomoeda
    const datasets = moedas.map((moeda, index) => ({
        label: `Fechamento ${moeda}`,
        data: historicoPorMoeda[moeda].map(d => d.fechamento),
        borderColor: cores[index % cores.length],
    }));

    const dashboard = new Chart(ctx, {
        type: 'line',
        data: {
            labels: labels,
            datasets: datasets //Vários datasets passados.
        },
    });
}
```

Você verá um resultado parecido com isso:

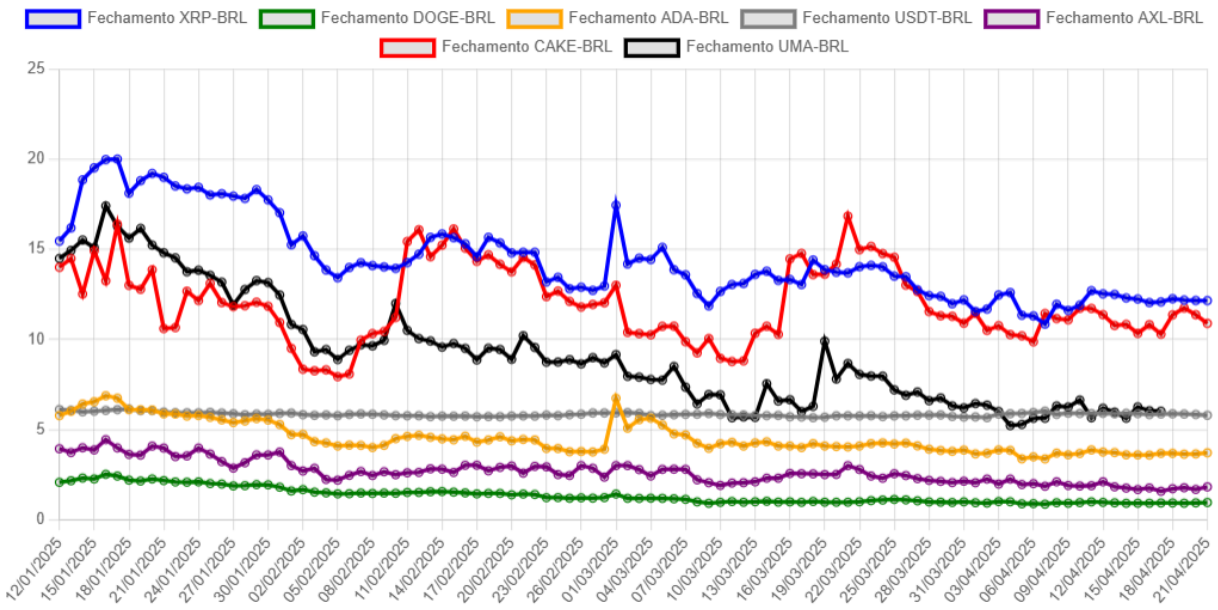


Figura 9.5: Dashboard de linhas com vários datasets

Observe que algumas criptomoedas são mais voláteis e oscilam bastante no preço, enquanto outras são mais estáveis e mantêm valores próximos durante um mesmo período. Esse é um *insight* que dificilmente perceberíamos apenas olhando uma planilha de dados. Com tantos números e datas, nossa mente facilmente se confundiria.

Já nesse Dashboard, conseguimos perceber essas variações "de imediato". Só de bater o olho, já identificamos quais criptos apresentam estabilidade e quais são mais inconstantes. Isso facilita — e muito — a tomada de decisão sobre comprar ou não determinado ativo, seja o comprador mais conservador ou adepto de estratégias de risco.

Assim como no Dashboard anterior, com poucas alterações você consegue adicionar ou remover comparações, ou ainda ajustar o período de tempo analisado. Para isso, basta incluir ou excluir itens no array `moedas` e modificar o valor da variável `dias` conforme a quantidade de dias que deseja comparar.

9.6 Prepare-se para o próximo nível

Neste capítulo, colocamos em prática tudo o que aprendemos até aqui. Construímos Dashboards reais, com diferentes formatos e fontes de dados. Vimos visualizações poderosas que facilitam a tomada de decisão.

Se você chegou até aqui, parabéns! Agora você já consegue montar Dashboards completos com HTML, JavaScript, Bootstrap e Chart.js.

Mas eu quero levar você para um próximo nível: criar soluções mais customizadas. Funcionalidades que não apenas melhoram a experiência de quem visualiza os Dashboards, mas também facilitam o trabalho de quem os constrói.

Sabemos que o tratamento de dados faz parte do trabalho antes da etapa visual. Deixamos isso claro no capítulo anterior, mas o JavaScript pode nos ajudar ainda mais, padronizando etapas recorrentes do processo de preparação dos dados.

Pensando nisso, no próximo capítulo vamos dar um passo além: criar nosso próprio importador de dados. A ideia é construir uma ferramenta simples e eficaz que transforma qualquer tabela de dados em um formato padrão, pronta para ser visualizada em Dashboards.

Vamos entender como construir uma base sólida e reutilizável para seus projetos. Juntos, vamos aprender a criar uma solução customizada que economiza tempo e oferece liberdade para trabalhar com qualquer fonte de dados.

Nos vemos no próximo capítulo!

Capítulo 10
Importador inteligente de dados

Neste capítulo, vamos abordar uma situação muito comum para quem trabalha com grande volume de dados.

Até aqui, vimos como integrar APIs (e que o JavaScript lida muito bem com isso), aprendemos a montar telas para inserir dados manualmente e a projetar visualmente nossos Dashboards. Isso tudo funciona muito bem, mas e se o volume de dados for grande?

Mesmo que seja fácil montar um formulário para preencher na tela, quando os dados vêm em grande quantidade, não adianta: você vai acabar gastando um tempão digitando linha por linha. E, na prática, o que mais aparece são as famigeradas planilhas.

Às vezes, em uma reunião rápida, alguém abre o Excel ou o Google Sheets, joga ali umas estatísticas para comparar, "olha aqui o quanto aumentou isso, ou como caiu aquilo ali" e pronto. Pega dados de um sistema X, cruza com outro Y e começa a tirar conclusões ali mesmo. É direto e eficaz. Só que isso nos leva a duas limitações no que vimos até agora:

- Se esses dados forem criados na hora, ou extraídos manualmente, não haverá uma API pronta para seu JavaScript buscar;
- Mesmo que sejam poucos dados, montar uma tela personalizada para cada planilha nova leva tempo, e ainda exige que alguém digite todos os registros.

Aliás, foi justamente para resolver esse tipo de problema que surgiram soluções como o Power BI, da Microsoft, o Tableau e outras ferramentas de BI que permitem importar planilhas e gerar gráficos rápidos.

Mas há dois pontos importantes: essas ferramentas são pagas e não são acessíveis para todas as empresas (o Power BI, por exemplo, exige uma conta Microsoft Pro para uso completo). E o mais importante: você está aprendendo a construir seu próprio Dashboard, com sua lógica, com sua cara e do zero.

Então, neste capítulo, vamos criar um importador inteligente de dados.

A ideia é simples, mas poderosa: construir uma ferramenta própria que padronize a leitura dessas tabelas, transformando automaticamente cada registro em um objeto JavaScript pronto para uso. Com isso, você poderá:

- Reutilizar o mesmo código para qualquer tabela de dados;
- Criar Dashboards a partir de dados reais, mesmo que venham de arquivos variados;
- Ter um ponto de partida universal para os seus projetos.

Vamos começar!

10.1 Por que criar um importador?

Antes de mais nada, é importante saber que qualquer linguagem de programação lida bem com arquivos de texto. São dados simples, diretos, sem muitas camadas por trás. Lembra do formato JSON que estudamos lá na Parte II do livro? Ele foi criado com essa proposta: ser um jeito leve de representar dados em texto. Qualquer linguagem consegue entendê-lo.

Para fixar a ideia, vamos pensar no oposto: o que não seria um texto simples? Pense no Microsoft Word. Você digita um texto, mas aquele conteúdo contém uma série de informações extras: qual fonte está sendo usada, o tamanho da letra, se tem negrito, qual o espaçamento, o alinhamento, o posicionamento para impressão e

por aí vai. Por trás do texto que você está vendo, existe toda uma estrutura embutida que só faz sentido para o Word, e não para um programa de leitura de dados comum.

O mesmo ocorre com o Excel. Quando você salva uma planilha no formato `.xlsx`, está criando um arquivo que contém muito mais do que apenas os dados. Ele guarda bordas, cores, tamanhos de colunas, estilos de célula, formatações e várias outras camadas de configuração.

O JavaScript consegue acessar os dados dentro desse tipo de arquivo? Consegue, sim! Mas quanto mais camadas o arquivo tem, mais complexa se torna a leitura e o tratamento desses dados. Isso exige mais bibliotecas, mais código, mais previsões, e tudo isso vai contra a ideia de termos um importador leve e fácil de reaproveitar.

E aqui entra o nosso primeiro ponto-chave: é comum que os dados cheguem até nós em arquivos do Excel. Mas se queremos criar uma ferramenta simples e genérica, precisamos encontrar uma forma de ignorar as camadas de formatação e extrair apenas os dados puros. É aí que entra o formato `.csv`.

10.2 O formato CSV

Agora que entendemos que arquivos com muitas camadas de formatação dificultam a vida de quem programa, precisamos de um formato que vá na direção oposta: simples, leve e direto ao ponto. É exatamente isso que o CSV entrega.

CSV é a sigla em inglês para *Comma-Separated Values*, ou seja, valores separados por vírgulas. Ele é, literalmente, um arquivo de texto onde cada linha representa um registro e cada campo é separado por vírgula. Só isso.

Não há cor de célula, borda de tabela, tamanho da letra ou fórmulas. Apenas os dados.

Por exemplo, um arquivo `.csv` com nomes, idades e documento poderia ter este conteúdo:

```
nome,idade,documento
Lucas,35,1234-5
Priscilla,40,1122-3
Gabriel,30,9988-7
Luciana,50,7755-9
```

Repare que é praticamente o que faríamos no Bloco de Notas. E por isso mesmo, qualquer linguagem de programação consegue processar esse tipo de arquivo com facilidade, inclusive o JavaScript.

O Excel (e outros editores de planilhas) consegue exportar qualquer arquivo `.xlsx` para o formato `.csv`. Ao fazer isso, ele remove todas as camadas visuais (bordas, estilos etc.) e salva apenas o conteúdo da célula, separado por vírgula. É como se o Excel dissesse: "Está aqui só o seu dado cru, sem nenhum enfeite".

Além disso, o CSV é tão leve e universal que virou padrão na maioria dos sistemas que oferecem exportação de dados. Sistemas de gestão, bancos de dados, ERPs, CRMs e plataformas web geralmente têm um botão do tipo "Exportar para CSV". Isso quer dizer que a chance de você encontrar um sistema que ofereça exportação em CSV é muito maior do que encontrar um que já forneça uma API pronta para uso.

Isso é ótimo para nós, pois, se conseguimos padronizar a leitura de CSVs no nosso importador, estamos preparados para lidar com planilhas de praticamente qualquer sistema ou aquelas criadas em qualquer reunião (como já mencionamos) de maneira rápida, leve e reutilizável.

Como transformo a planilha em CSV?

Trazendo para a prática, haverá casos em que você vai ter que lidar com os arquivos de Excel ou até mesmo do Google Sheets, e esse formato CSV é tão universal que tanto a Microsoft quanto o Google disponibilizam a possibilidade de converter suas planilhas em CSV.

Vamos usar de exemplo esses mesmos dados, porém em um arquivo Excel chamado `lista.xlsx`:

	A	B	C
1	**nome**	**idade**	**documento**
2	Lucas	35	1234-5
3	Priscilla	40	1122-3
4	Gabriel	30	9988-7
5	Luciana	50	7755-9

Figura 10.1: Exemplo de dados em Excel

Note que é um arquivo com mais camadas — títulos em negrito, bordas, alinhamentos. No próprio Excel, no menu principal "Arquivo", existe uma opção "Salvar como", em que você digita o nome do arquivo e, logo depois, há um campo com os tipos suportados. Haverá uma lista de opções — entre elas, você vai ver o CSV. Salve em CSV e você terá o arquivo com os dados de forma tabelada, mas em texto simples.

E se a planilha estiver online, no Google Sheets? Também haverá essa opção:

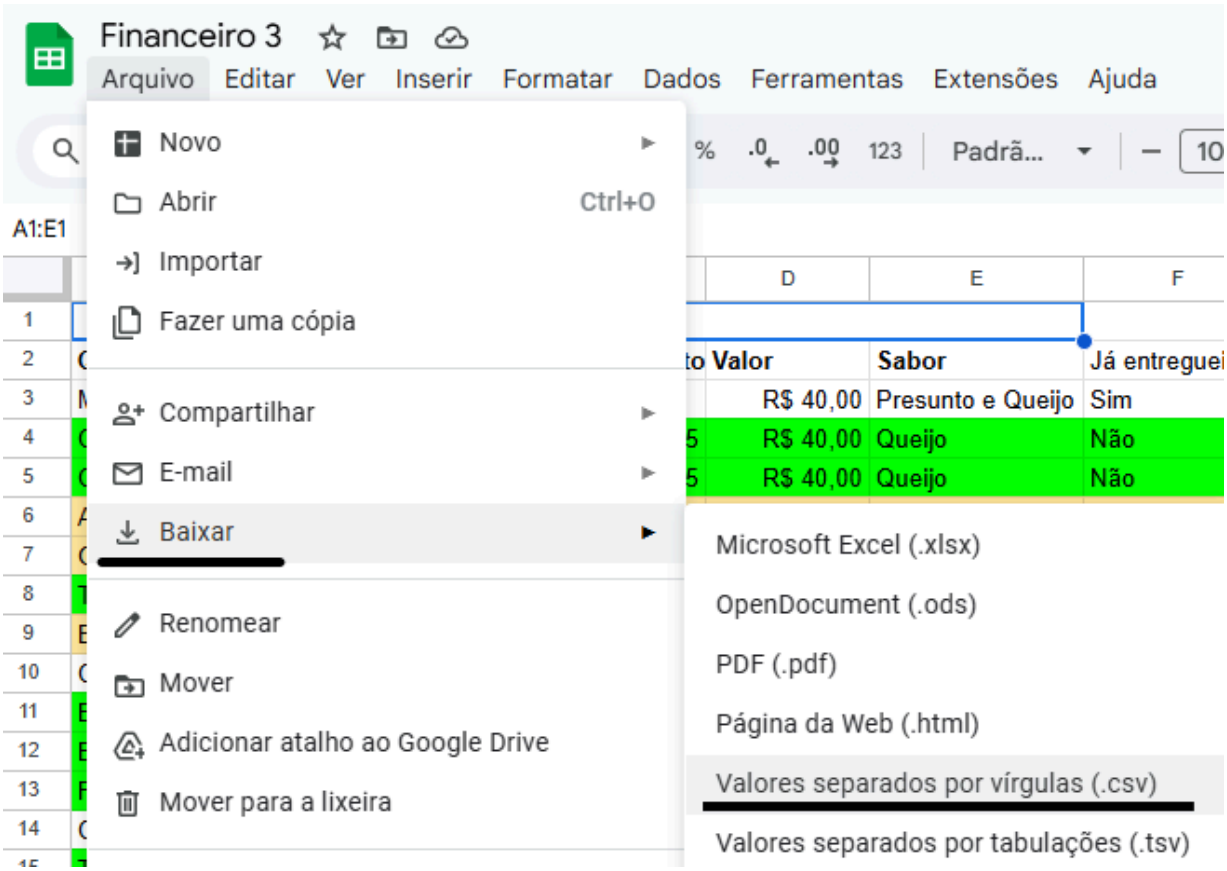


Figura 10.2: Exemplo de dados do Google Sheets

Pensando no fluxo geral, chegamos a uma forma de sempre trabalhar com os dados limpos, para que, no próximo passo, nosso importador organize tudo para nós.

10.3 Passo a passo do importador

Para criarmos nosso importador, precisamos ter uma página que permita a inserção do arquivo `.csv`. Você pode criar um HTML com mais funções e implementar descrições e funcionalidades como preferir. Para a funcionalidade de importar esses arquivos, nós só precisamos de um `<input>` do tipo `file` para receber um arquivo.

Algo assim:

```
<!-- Seu HTML -->
<input type="file" id="inputArquivo" accept=".csv"
onchange="lerArquivo()">
```

Esse `input` criará um campo na sua tela que permite a inserção de arquivos. O `id="inputArquivo"` identifica esse campo para o JS conseguir pegar o arquivo. O `accept=".csv"` faz com que o seu HTML permita apenas arquivos do tipo CSV e o `onchange="lerArquivo()"` é a função acionada no JavaScript. Nela, estarão todos os mecanismos do nosso importador. Vamos ver essa função detalhadamente:

```
function lerArquivo() {
    const input = document.getElementById('inputArquivo'); //Campo de
upload do arquivo
    const file = input.files[0]; //Pega o primeiro arquivo da lista
(normalmente usamos apenas um arquivo por vez)
    if (!file) return alert('Nenhum arquivo selecionado.'); //Verifica se
o arquivo foi selecionado

    const leitor = new FileReader(); //Cria um novo objeto FileReader para
ler o arquivo
    const extensao = file.name.split('.').pop().toLowerCase(); //Pega a
extensão do arquivo e transforma em minúscula

    leitor.readAsText(file); //Faz a leitura do arquivo como texto
    leitor.onload = function (e) { //Aciona quando o arquivo for lido
        let resultado; //Variável para guardar o resultado

        const linhas = e.target.result.trim().split('\n'); //Lê todo o
conteúdo do arquivo e separa as linhas - \n é o caractere de nova linha
        const colunas = linhas[0].split(','); //Pega a primeira linha do
arquivo, que contém os nomes das colunas, utilizando a vírgula como
separador. Lembre-se que pode haver planilhas de dados que usam outro tipo
de separador. Pode ser ponto e vírgula, tabulação etc. Para isso, é só
trocar a vírgula por outro caractere.

        resultado = linhas.slice(1).map(linha => { //Inicia a leitura de
todas as linhas, exceto o cabeçalho por isso começa na linha 1
            const valores = linha.split(','); //Pega os valores de cada
```

```
linha, utilizando a vírgula como separador - Lembre que pode ser outro
caractere
            const obj = {}; //Cria um objeto vazio para armazenar os
valores
            colunas.forEach((chave, i) => { //A chave de cada objeto vem
do nome daquela coluna
                obj[chave.trim()] = valores[i].trim(); //A chave é o nome
da coluna e o valor é o valor daquela linha
            });
            return obj; //Retorna o objeto com todos os valores formatados
em array de objetos dentro da variavel "resultado"
        });

        console.log(resultado); //Mostra o resultado no console para
vermos os dados todos formatados.
    };
}
```

Ao executar esta tela, o `console.log(resultado);` presente na última linha do código vai exibir os dados trabalhados. Qualquer CSV que você importar usando este programa mostrará, no console, algo parecido com isto:

```
▼ (4) [{…}, {…}, {…}, {…}] i
  ▶ 0: {nome: 'Lucas', idade: '35', documento: '1234-5'}
  ▶ 1: {nome: 'Priscilla', idade: '40', documento: '1122-3'}
  ▶ 2: {nome: 'Gabriel', idade: '30', documento: '9988-7'}
  ▶ 3: {nome: 'Luciana', idade: '50', documento: '7755-9'}
    length: 4
  ▶ [[Prototype]]: Array(0)
```

Figura 10.3: Resultado no console

IMPORTANTE! Nos trechos `linhas[0].split(',')` e `linha.split(',')`, a separação dos dados está sendo feita pelo caractere `,`. Isso significa que, para funcionar corretamente, o arquivo CSV precisa usar a vírgula como separador. Se o arquivo `.csv` usar outro caractere como separador, basta substituir o caractere nesses trechos por aquele que estiver sendo usado. Por exemplo, o caractere `;` também aparece como separador de informação, então é preciso alterar para `linhas[0].split(';')` e `linha.split(';')`.

Exemplo prático

```
//Se seu CSV usa ponto e vírgula como separador:
const colunas = linhas[0].split(';');
const valores = linha.split(';');
```

Essa é a função que vai facilitar nossa vida daqui para a frente. Será o nosso importador padrão. Ela pega os dados de qualquer arquivo CSV e deixa estruturado em um array para nós. Com esse array de objetos em mãos, agora temos uma base limpa, padronizada e fácil de manipular, pronta para alimentar qualquer Dashboard que quisermos criar.

DICA EXTRA

No capítulo 8, falamos da criação de um arquivo de scripts `utilitarios.js` para organizar funções úteis que podem ser utilizadas em diversos casos. A função `lerArquivo()` se aplica a este caso. Ela será usada diversas vezes. Você pode até fazer duas versões dela com adaptação para o caractere separador. Por exemplo: função `lerArquivoApenasVirgula()` para arquivos com separador `,` e a função `lerArquivoPontoeVirgula()` para arquivos com separador `;`.

Neste ponto do livro, já exploramos como arrays de objetos são estruturas ideais para trabalhar com dados em JavaScript, especialmente quando usamos métodos como `.map()`, `.filter()` e `.reduce()`. Veja que isso é um passo além: transformar uma planilha nesse formato. A ideia é que cada linha da planilha se torne um objeto, e cada coluna, uma chave desse objeto. Esse processo de conversão facilita muito a manipulação dos dados e a criação de Dashboards dinâmicos.

10.4 Dashboards com o importador

Chegou o momento perfeito para colocar o nosso importador inteligente de dados em ação. Vamos montar Dashboards reais com base nos dados que você carregou e organizou por meio do importador. Vamos partir do princípio de que o seu importador já está funcionando, e que você já tem um array chamado `resultado` com os dados formatados.

> Lembre-se de que todos os exemplos deste livro estão disponíveis no repositório oficial: https://github.com/luctauil/livro_dashboards_com_js.git.

Para este exemplo, vamos usar um arquivo chamado `vendas_vendedores.csv` com um conteúdo parecido com este:

```
mes,vendedor,vendas
2024-11,Ana,19697
2024-11,Bruno,16858
2024-11,Carla,19358
2024-11,Daniel,10909
```

```
2024-11,Eduarda,19607
...
```

Essas são apenas as primeiras linhas do arquivo. O arquivo completo está disponível no repositório caso queira utilizar com mais dados.

Após carregar o arquivo usando o importador, seu JavaScript terá transformado os dados em um array de objetos assim:

```
{mes: '2024-11', vendedor: 'Ana', vendas: '19697'}
{mes: '2024-11', vendedor: 'Bruno', vendas: '16858'}
{mes: '2024-11', vendedor: 'Carla', vendas: '19358'}
{mes: '2024-11', vendedor: 'Daniel', vendas: '10909'}
{mes: '2024-11', vendedor: 'Eduarda', vendas: '19607'}
...
```

Esses também são apenas os primeiros registros. Assim, o importador terá formatado todos os registros existentes no arquivo CSV.

Esse formato é ideal porque já permite trabalhar diretamente com funções como `map`, `filter` e `reduce`, além de ser compatível com bibliotecas como Chart.js.

Vamos criar dois Dashboards com o mesmo conjunto de dados:

- Um comparativo entre vendedores por mês (Dashboard de linha);
- Um totalizador geral de vendas por vendedor (Dashboard de barras).

Para isso, precisaremos de dois elementos `canvas` no HTML:

```
<div>
    <canvas id="meu_dashboard"></canvas>
</div>

<div>
    <canvas id="meu_dashboard2"></canvas>
</div>
```

> **LEMBRETE!** Você pode posicionar esses elementos como preferir: um abaixo do outro, lado a lado ou dentro de abas. O importante é que ambos existam para que possamos renderizar os dois Dashboards.

Agora, adicionamos um botão para gerar:

```
<div>
    <button onclick="CarregarDashboards()">Gerar Dashboard</button>
</div>
```

Este botão executará a função `CarregarDashboards()` , que por sua vez chama outras duas funções responsáveis por cada Dashboard:

```
function CarregarDashboards() {
    GerarDashboardVendedores(resultado); //Neste vamos fazer o comparativo entre vendedores
    GerarDashboardTotais(resultado); //Neste vamos fazer o totalizador de vendedores
}
```

Comparativo entre vendedores no período

Uma das visualizações mais úteis neste tipo de dado é comparar o desempenho de cada vendedor ao longo dos meses. O gráfico de linha é ideal para esse tipo de análise: com ele, conseguimos enxergar, em uma única visualização, quais vendedores mantêm uma constância, quais se destacam e até momentos de queda ou pico nas vendas. Uma visualização, várias informações.

A estrutura do Dashboard será baseada nas seguintes relações:

```
const meses = dados.map(d => d.mes).filter((mes, i, self) => self.indexOf(mes) === i);
const vendedores = dados.map(d => d.vendedor).filter((v, i, self) => self.indexOf(v) === i);
const cores = ['darkblue', 'pink', 'blue', 'black', 'orange', 'lightgreen', 'purple', 'darkgreen', 'gray'];
```

Com essas variáveis do tipo arrays, conseguimos construir os eixos e as séries do Dashboard, associando uma linha a cada vendedor com uma cor distinta. A partir disso, é só gerar os dados de vendas de cada mês a cada vendedor:

```
// Para cada vendedor identificado...
for (let i = 0; i < vendedores.length; i++) {
    const vendedor = vendedores[i];

    // Para cada mês, vamos verificar se há um registro de venda para este vendedor.
    // Se houver, pegamos o valor da venda; se não, colocamos 0 para manter a estrutura do Dashboard.
    const vendasPorMes = meses.map(mes => {
        const item = dados.find(d => d.vendedor === vendedor && d.mes === mes);
        return item ? parseInt(item.vendas) : 0;
    });

    // Montamos o dataset individual e adicionamos ao array final
    datasets.push({
        label: vendedor, // nome do vendedor na legenda
        data: vendasPorMes, // array com os valores mês a mês
        borderColor: cores[i % cores.length], // cor diferente para cada linha
    });
}
```

Por fim, com os `datasets` prontos, basta colocarmos em um Dashboard de linhas como fizemos lá no primeiro exemplo do capítulo anterior:

```
const ctx = document.getElementById('meu_dashboard').getContext('2d');
const dashboard = new Chart(ctx, {
    type: 'line',
    data: {
        labels: meses,
        datasets: datasets
```

```
    },
});
```

O resultado será:

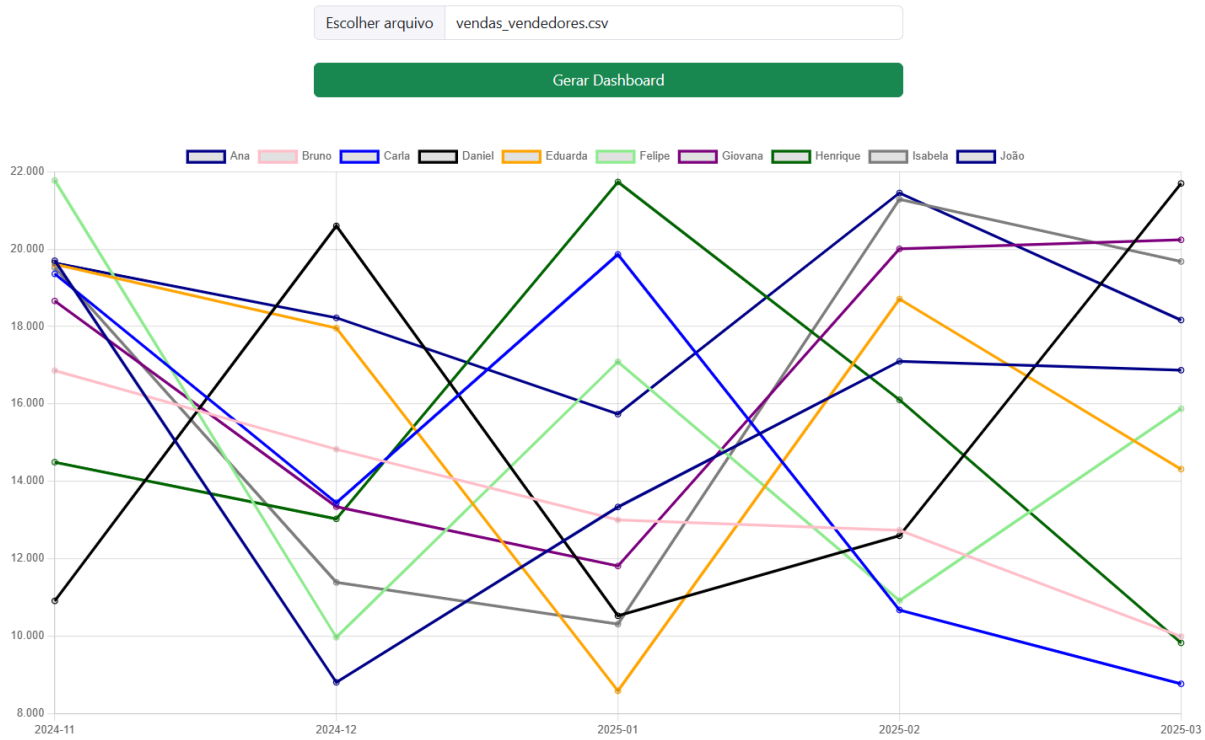


Figura 10.4: Comparativo entre vendedores

Totalizador geral de vendas no período

O segundo Dashboard que vamos montar a partir do mesmo array `resultado` é um totalizador geral de vendas por vendedor. Enquanto no Dashboard anterior mostramos a evolução mês a mês, aqui o foco é mostrar quem vendeu mais no geral, somando todos os meses e exibindo isso em um Dashboard de barras.

Primeiro, montamos a lista única de vendedores:

```
const vendedores = dados.map(d => d.vendedor).filter((v, i, self) =>
self.indexOf(v) === i);
```

Para cada vendedor, vamos somar todas as suas vendas, independentemente do mês:

```
const totais = [];

// Para cada vendedor...
for (let i = 0; i < vendedores.length; i++) {
    const vendedor = vendedores[i];

    // ...filtramos todos os registros dele
    const vendasDoVendedor = dados.filter(d => d.vendedor === vendedor);

    // ...somamos todas as vendas usando reduce
    const total = vendasDoVendedor.reduce((acc, atual) => acc +
parseFloat(atual.vendas), 0);

    // ...e adicionamos ao array totais
    totais.push(total);
}
```

Observe que temos um array com os vendedores e outro com o total de vendas de cada um. Com isso, podemos montar um Dashboard de barras vertical como este passando a variável `totais` como fonte dos dados:

```
const ctx = document.getElementById('meu_dashboard2').getContext('2d');
new Chart(ctx, {
    type: 'bar',
    data: {
        labels: vendedores,
        datasets: [{
            label: 'Total Geral por Vendedor',
            data: totais,
            backgroundColor: "darkgreen"
        }]
    }
});
```

O resultado será algo como:

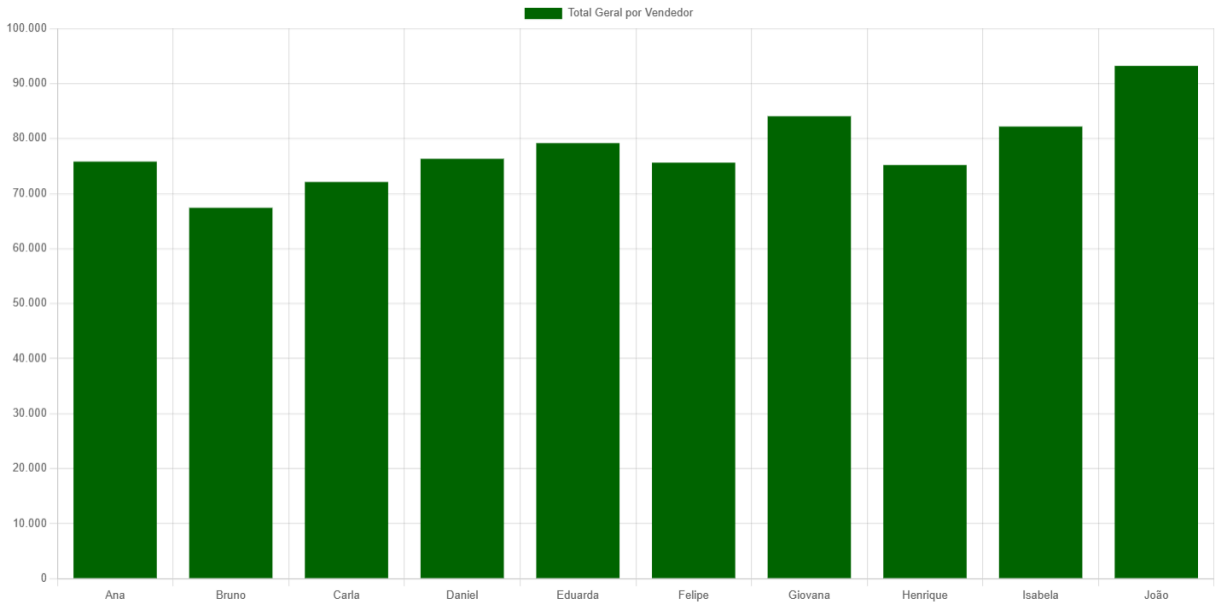


Figura 10.5: Totalizador de vendas por vendedor

Com o que você aprendeu neste capítulo, deixou de ser apenas um consumidor de dados prontos e passou a criar sua própria ferramenta de importação: um pequeno sistema adaptável, capaz de trabalhar com praticamente qualquer planilha `.csv`, seja ela produzida em reuniões rápidas ou exportada de sistemas mais antigos.

Você viu como é possível importar arquivos, extrair os dados em um formato padronizado (array de objetos) e, a partir disso, gerar múltiplos Dashboards diferentes com facilidade. Em outras palavras, você acabou de montar seu próprio motor de análise de dados.

Esse importador inteligente não serve só para este livro. Ele pode ser o ponto de partida para futuros projetos, relatórios, Dashboards, soluções de BI personalizadas, sistemas de acompanhamento e muito mais.

No próximo capítulo, vamos reunir tudo o que vimos até aqui para montar um super Dashboard, incluindo elementos visuais mais avançados e uma visualização inédita: um mapa do Brasil com dados por estado.

Capítulo 11
Visualizando o Brasil: Dados em mapa

Como vimos no final do capítulo anterior, neste ponto do livro você já domina todas as ferramentas necessárias para criar seus próprios Dashboards — do início ao fim, do básico ao avançado.

A proposta deste capítulo é mostrar que, com o conhecimento que você adquiriu, não estamos mais limitados a ferramentas prontas ou tutoriais específicos. Agora é uma questão de criatividade: vamos combinar técnica com dados disponíveis na internet e construir um Dashboard totalmente personalizado.

Neste exemplo, usaremos um mapa do Brasil como base, aproveitando a riqueza de informações públicas sobre os estados. Mas, no seu caso, o tema pode ser outro. Algo exclusivo de um setor, um gasto mal interpretado, uma tendência ainda não percebida. Pode ser qualquer informação que só você enxergou como relevante e que precisa ser visualizada para ganhar força.

Começaremos estruturando o mapa e, em seguida, buscaremos dados dispersos para transformá-los em uma visualização que surpreenda, algo que ninguém havia notado até agora.

11.1 Exibindo mapas com Leaflet.js

O primeiro passo é criar o mapa. Assim como exposto em outras situações ao longo do livro, podemos utilizar uma biblioteca pronta para facilitar esse processo. Existem várias opções no mercado, sendo que uma das mais conhecidas é o Google Maps, que oferece uma infinidade de funcionalidades robustas e disponíveis para uso. No entanto, a versão gratuita possui limitações e, para quem precisa de algo mais avançado, pode ser necessário contratar a versão paga.

Como a proposta aqui é trabalhar com ferramentas gratuitas e acessíveis, utilizaremos o *Leaflet*, que é uma biblioteca JavaScript *open source* para criação de mapas interativos. Ela é leve, moderna e fácil de usar. Mesmo sendo gratuita, oferece recursos poderosos como zoom, camadas de informação, ícones personalizados e compatibilidade com diversos serviços de mapas (como *OpenStreetMap*). Por isso, tornou-se uma das bibliotecas mais populares para visualização geográfica na web.

Para usá-lo, precisamos apenas adicionar duas linhas no cabeçalho do nosso HTML:

```
    <link rel="stylesheet"
href="https://unpkg.com/leaflet/dist/leaflet.css" />
    <script src="https://unpkg.com/leaflet/dist/leaflet.js"></script>
```

Essas linhas trazem os arquivos principais da biblioteca diretamente da web, sem precisar instalar nada.

> Se você quiser conhecer todos os recursos que o *Leaflet* oferece, vale a pena explorar a documentação oficial da biblioteca. Ela é bem organizada, com exemplos práticos e tutoriais para diversos níveis de conhecimento. Acesse em: https://leafletjs.com/.

Com o *Leaflet* carregado na página, podemos criar o nosso mapa usando o seguinte código HTML e JavaScript. Ele já está centralizado no Brasil e com um nível de zoom adequado para visualização geral dos estados.

```
<div id="map" style="width: 100%; height: 350px;"></div>
```

Essea div com `id="map"` é onde o *Leaflet* renderizará o mapa. O estilo `width: 100%; height: 350px;` garante que ele ocupe toda a largura disponível e tenha uma altura visível adequada na tela.

Lembre-se de criar o seu arquivo JS e apontá-lo no HTML. Pode ser assim:

```
<script src="scripts/mapa.js"></script>
```

Neste seu arquivo JavaScript, basta adicionar:

```
onload = () => {
    // Inicializa o mapa dentro da div com id 'map'
    // O método setView recebe dois parâmetros:
    // - Coordenadas (latitude e longitude) para centralizar o mapa
    // - Nível de zoom (quanto menor, mais distante)
    const map = L.map('map').setView([-14.2350, -51.9253], 3); //
Centralizado no Brasil

    // Adiciona o conjunto de tiles (camadas visuais) ao mapa
    // Este exemplo usa o OpenStreetMap, uma fonte gratuita e aberta
    L.tileLayer('https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png', {
        maxZoom: 20, // Zoom máximo permitido
    }).addTo(map);
};
```

O código JavaScript segue a estrutura padrão de inicialização recomendada pela própria documentação do *Leaflet*. A biblioteca em si cuida da parte interativa do mapa (como zoom, movimentação e camadas), mas ela não fornece os "tiles", ou seja, as imagens que compõem o mapa. Para isso, usamos um link do provedor gratuito *OpenStreetMap*, que é compatível com o Leaflet e muito utilizado em projetos públicos e privados.

A combinação do *Leaflet* com *tiles* do *OpenStreetMap* é uma escolha popular por ser leve, gratuita e fácil de integrar em projetos web.

Se quiser personalizar a visualização inicial do mapa, você pode alterar os dois parâmetros do método `setView`. O primeiro define as coordenadas geográficas em formato `[latitude, longitude]` (por exemplo, `[-23.5505, -46.6333]` para São Paulo). O segundo parâmetro define o nível de zoom, sendo que valores mais baixos mostram áreas mais amplas e valores mais altos aproximam mais a visualização. Você também pode ajustar o `maxZoom` na configuração do `tileLayer` para limitar o quanto o usuário pode se aproximar do mapa.

Ao executar este código, você verá algo como:

Figura 11.1: Mapa do Brasil em JS

11.2 Integrando GeoJSON ao Leaflet

Até aqui, criamos o nosso mapa e o centralizamos no Brasil. Mas como podemos destacar os estados e associar informações a eles? É aí que entra o **GeoJSON**.

O GeoJSON é um formato de arquivo baseado em JSON que representa elementos geográficos como pontos, linhas e polígonos. No nosso caso, usaremos um arquivo `.json` contendo o contorno de todos os estados do Brasil. Cada estado é representado como um **polígono**, com suas coordenadas e metadados (nome, sigla, entre outros).

Esses arquivos são gratuitos e podem ser encontrados em diversas fontes confiáveis como:

- **IBGE** (https://www.ibge.gov.br/) – oferece arquivos de malha territorial atualizados e detalhados.
- **GeoJSON.xyz** (https://geojson.xyz) – uma galeria pública de arquivos GeoJSON prontos para uso.
- **DataHub** (https://datahub.io/) – repositório de dados abertos, com boa curadoria.
- **Repositórios no GitHub** – muitos desenvolvedores já publicaram versões prontas da divisão por estados ou municípios.

Outra opção é gerar seu próprio GeoJSON usando ferramentas como o editor online **geojson.io** (https://geojson.io), que permite desenhar polígonos diretamente no mapa e exportar o arquivo no formato adequado. A cada clique no mapa, o seu GeoJSON vai sendo construído conforme o desenho. Depois, é só salvar o arquivo.

Você deve estar pensando: "Quer dizer que eu teria que desenhar todo o contorno do Brasil, estado por estado, só para gerar meu próprio GeoJSON?" Seria um trabalhão, não é? Mas fique tranquilo(a), existem desenvolvedores que já fizeram isso por você e liberaram esses arquivos gratuitamente para a comunidade. É o caso de repositórios como:

- https://github.com/giuliano-macedo/geodata-br-states
- https://github.com/tbrugz/geodata-br

Mais especificamente, vamos usar o arquivo `geojson/br_states.json`, disponível no repositório do Giuliano Macedo. Minha sugestão é que você baixe esse arquivo e coloque-o em uma pasta do seu projeto, por exemplo: `/dados/br_states.json`. A partir disso, basta carregá-lo no JavaScript com o `fetch`.

Para que seu código já comece com ele carregado, você pode colocá-lo junto do `onload` anterior, desta forma:

```
onload = () => {
    const map = L.map('map').setView([-14.2350, -51.9253], 3);
    L.tileLayer('https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png', {
```

```
        maxZoom: 20,
    }).addTo(map);

    //Carregando o GeoJSON
    fetch('../../dados/br_states.json') //Coloque o caminho que você
salvou.
        .then(response => response.json())
        .then(data => {
            console.log(data); //Carregado na variável data.
        });
}
```

Esse código vai buscar o arquivo `br_states.json` que você deve colocar dentro do seu projeto. Ao abrir o navegador e recarregar a página, abra o console e você verá a estrutura completa do arquivo sendo exibida. Você encontrará todos os dados geográficos já organizados em um array de objetos, com a imagem a seguir:

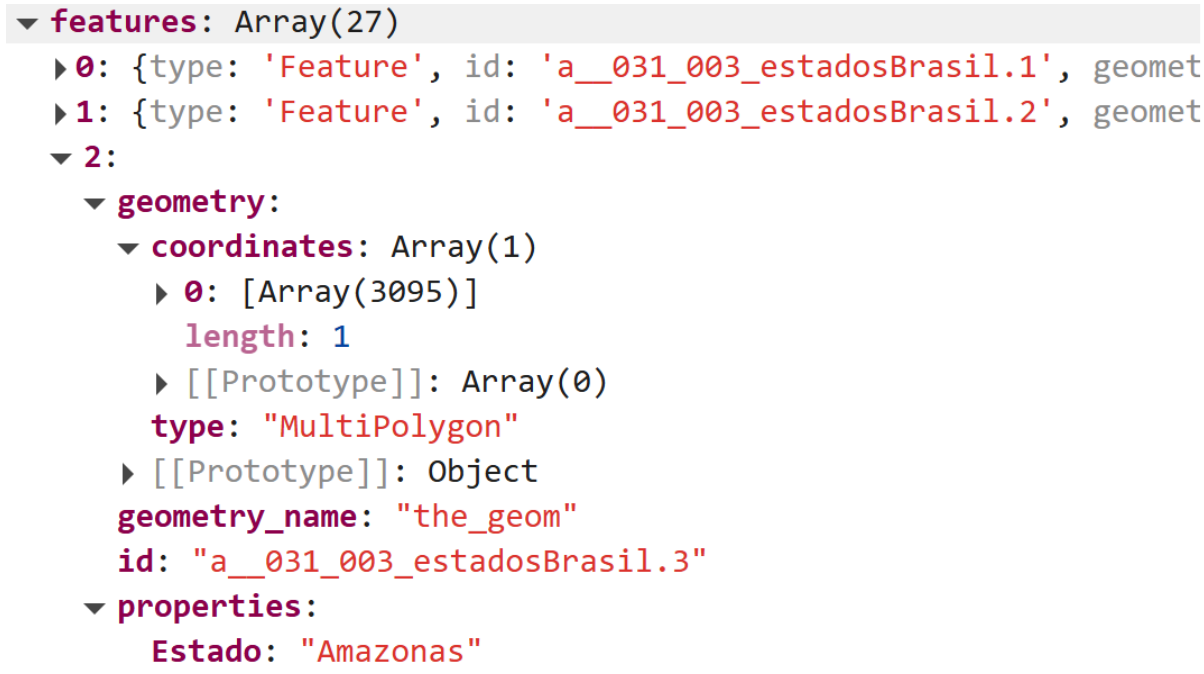


Figura 11.2: GeoJSON

Lembra que citamos vários benefícios de usar o array de objetos? Aqui está mais um exemplo! A biblioteca de mapas *Leaflet* já está preparada para receber GeoJSONs neste formato, e conseguimos

renderizar todas as coordenadas deste array no nosso mapa adicionando o seguinte código:

```
fetch('../../dados/br_states.json')
    .then(response => response.json())
    .then(data => {
        L.geoJSON(data).addTo(map); //Esse comando adiciona os dados no
map
});
```

E o que antes era um simples mapa "cru" do Brasil na tela agora ganha destaque com contornos bem definidos de cada estado.

Você perceberá um destaque azul nas regiões: essa é a cor padrão utilizada quando não há personalização. Mudaremos isso adicionando mais alguns parâmetros diretamente no `L.geoJSON`. Essa função aceita um segundo argumento com várias características visuais, como neste exemplo:

```
L.geoJSON(data, {style: 'array_de_estilos_aqui' } ).addTo(map);
```

Vamos criar uma função para facilitar a organização deste objeto de estilos:

```
function Estilos() {
    return {
        fillColor: 'green', // Cor interna dos estados
        weight: 2,          // Espessura do contorno
        opacity: 1,         // Opacidade do contorno
        color: 'black',     // Cor da linha do contorno
        fillOpacity: 0.7    // Transparência do preenchimento
    };
}
```

Fique à vontade para testar outras cores e ver como o seu mapa ganha vida. Depois de criar a função, basta chamá-la assim:

```
L.geoJSON(data, {style: Estilos } ).addTo(map);
```

Todo o código de carregamento do mapa ficará assim:

```
onload = () => {
    const map = L.map('map').setView([-14.2350, -51.9253], 3);
    L.tileLayer('https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png', {
        maxZoom: 20,
    }).addTo(map);

    fetch('../../dados/br_states.json') //Seu arquivo pode estar em outro
caminho
        .then(response => response.json())
        .then(data => {
            L.geoJSON(data, { style: Estilos } ).addTo(map);
        });
}
```

Com isso, temos um mapa do Brasil carregado na tela, com contorno e preenchimento colorido para cada estado. Essa é a base visual que usaremos a partir de agora.

O mapa que aparece na tela é renderizado com base nas informações do nosso GeoJSON, e vinculamos esses dados à função `Estilos` que criamos. Para mostrar o nível de personalização que já conseguimos alcançar, proponho uma alteração simples nessa função:

```
function Estilos(feature) {
    const estado = feature.properties.SIGLA;

    let cor = 'green';
    if (estado === 'SP') cor = 'blue';
    else if (estado === 'RJ') cor = 'red';
    else if (estado === 'MG') cor = 'orange';

    return {
        fillColor: cor,
        weight: 2,
        opacity: 1,
        color: 'black',
        fillOpacity: 0.7
```

```
    };
}
```

A adaptação mostrada faz uso do parâmetro `feature`, que é fornecido automaticamente pelo *Leaflet* para cada item (ou estado) do GeoJSON. A partir dele, podemos acessar as propriedades de cada estado, como sua sigla, e definir cores diferentes de acordo com alguma condição. Ao atualizar a função, você verá cores distintas aplicadas a estados específicos, conforme as regras definidas.

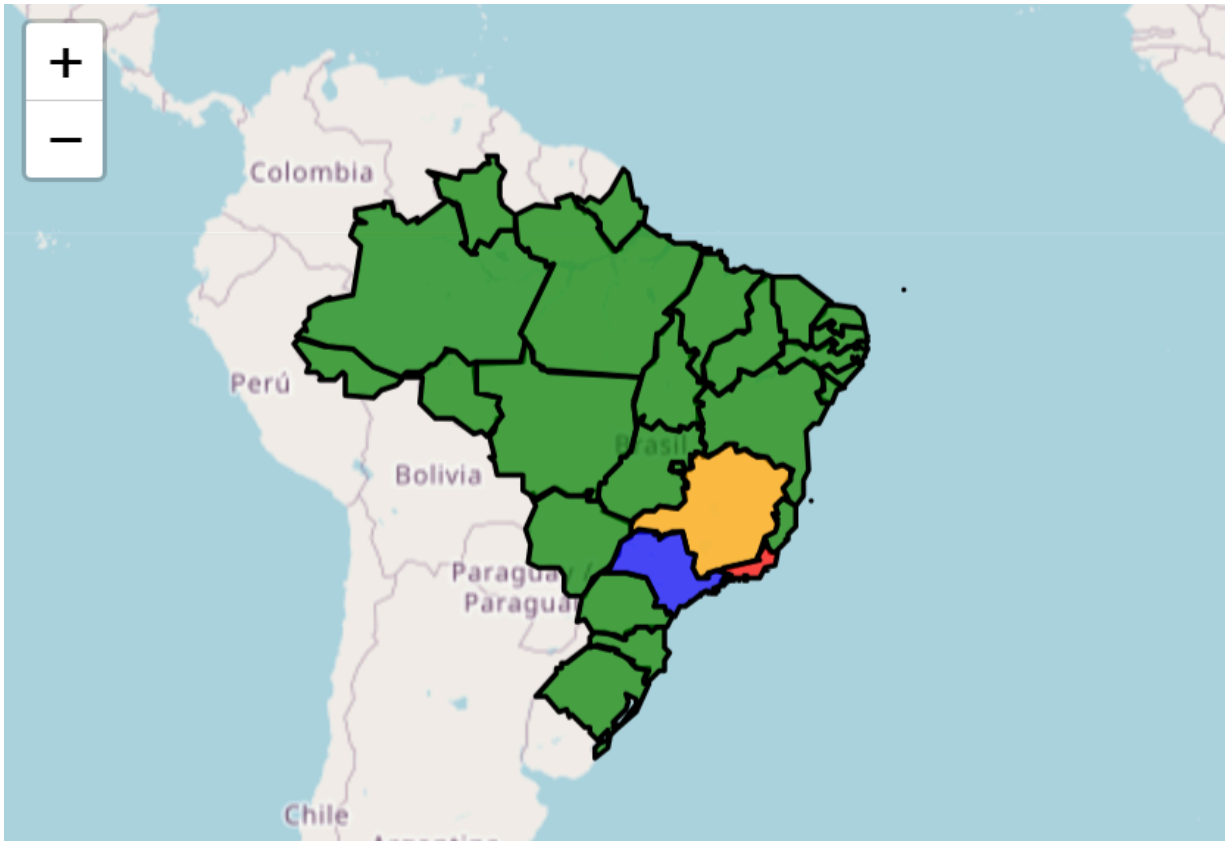


Figura 11.3: Mapa do Brasil por estado

Essa mudança serve apenas como exemplo inicial. Nas próximas seções, vamos integrar dados reais ao nosso projeto para destacar diferenças e padrões entre os estados. Esse mapa evoluirá e se transformará em um verdadeiro Dashboard interativo e colorido.

Mas antes de pintarmos esse mapa com novos dados, precisamos entender quais informações usaremos, e como elas serão organizadas para tornar o visual ainda mais expressivo.

Portanto, salve seu código e vamos falar um pouco sobre os dados.

11.3 Importando dados públicos para o Dashboard

Como estamos construindo um projeto de Dashboard ambicioso, que envolve pintar um mapa de todo o país, precisamos de informações que também cubram todo o território nacional. O exemplo perfeito para isso são os dados abertos da nossa federação. O Governo Federal disponibiliza um portal de dados abertos, em http://dados.gov.br/. Nesse site, você encontra diversos bancos de dados com informações reais e relevantes sobre o Brasil. Fica até como dica: esse portal é excelente para treinar, já que oferece conjuntos de dados com volumes expressivos, perfeito para quem deseja praticar a criação de Dashboards.

Dentre as várias bases disponíveis, uma que se destaca por ser fácil de entender (e muito interessante, já que lida com valores financeiros) é o banco de dados "Resultado da Arrecadação". Para encontrá-lo, digite no campo de pesquisa: "receita por estado". Depois clique em buscar. Se aparecer mais de um resultado, procure pelo banco de dados chamado "Resultado da Arrecadação". Ou, se preferir ir direto ao ponto, acesse: https://dados.gov.br/dados/conjuntos-dados/resultado-da-arrecadacao.

Você verá uma seção chamada "Recursos". Dentro dela, estão disponíveis várias formas de download e, claro, a nossa favorita está lá: o formato `.csv`. Após o download, mesmo que você abra o arquivo no Bloco de Notas, verá algo parecido com isso:

```
Ano;Mes;UF;IMPOSTO SOBRE IMPORTACAO;IMPOSTO SOBRE EXPORTACAO;IPI - FUMO;IPI - BEBIDAS;IPI - AUTOMOVEIS;IP
2000;Janeiro;AC;231;0;292096;0;0;167;1558;177197;643;676835;1084940;27652;570;30894;1220;28447;0;103;1770
2000;Janeiro;AL;475088;33873;1329338;812470;0;141735;3676847;460804;561;2672106;4454947;544056;191382;309
2000;Janeiro;AM;11679405;0;1507146;1791471;27796;4414483;1800346;700623;90828;10058713;10174718;1241708;3
2000;Janeiro;AP;24267;0;424862;1419;321;22333;12165;139226;44135;586572;1444486;205031;5825;34602;6999;64
2000;Janeiro;BA;10476457;13;5542123;11533707;0;7496476;23743858;2498647;177378;25695696;27277025;13374802
2000;Janeiro;CE;5576921;0;4535653;3148254;0;1734529;3716458;1971500;59874;13247447;15940490;6311419;94860
2000;Janeiro;DF;2254341;43;3197500;2777942;0;2079272;2433339;3067374;3913461;71582680;257391352;124316928
2000;Janeiro;ES;31950799;0;3598736;20408;1503039;26840430;4036463;1620843;2542;29105465;9380778;10515327;
2000;Janeiro;GO;3747911;0;3236092;5064141;1091185;986314;2654994;1401385;45754;18823998;13094463;6275215;
2000;Janeiro;MA;1714749;69;1124738;4061569;0;206691;484436;479170;4832;3251185;4149532;781569;1130;289120
2000;Janeiro;MG;35549181;57;15511142;12111623;29305691;15049305;38667924;11038328;3416372;83378349;734109
2000;Janeiro;MS;437668;0;899850;268111;0;365983;626188;1926537;7972;3983857;4331201;797218;26225;194378;5
2000;Janeiro;MT;302002;0;1837228;2734966;0;98807;322126;537577;5356;5998492;6702247;506327;287788;518377;
2000;Janeiro;PA;1321715;0;2506107;1255572;0;771308;842141;885476;1687010;10397922;9891932;1452409;271488;
2000;Janeiro;PB;1351422;0;1971817;5556557;0;763383;1729472;531092;6819;8455014;9164933;1315192;215491;193
2000;Janeiro;PE;6268803;13;6167644;5650794;0;2688235;8230496;1679485;245392;15783042;26054337;38164839;98
2000;Janeiro;PI;382374;0;1001688;2261830;0;88220;261477;291999;16605;4285628;3839267;1119727;58819;165302
2000;Janeiro;PR;45984119;0;10114872;5045577;12768408;23702615;26596165;5748682;3925211;60829302;39875650;
2000;Janeiro;RJ;54679975;110;24542326;30839175;4605;28145513;47564082;36256377;19404254;181047892;2472910
2000;Janeiro;RN;997074;0;1849649;1973085;0;586673;150429;493904;4783;3791527;9790539;3138513;201215;45201
2000;Janeiro;RO;10184;0;604552;0;0;6453;26622;703484;0;2405470;7955556;377430;2316;75311;7432;3955659;0;3
2000;Janeiro;RR;2047;0;248589;0;0;690;0;84094;2223;668266;1178651;72674;11610;24652;0;13251;0;4554;123585
```

Figura 11.4: Dados brutos por estado

O conteúdo estará no formato que esperamos de qualquer arquivo `.csv` : a primeira linha contém os nomes das colunas, e as linhas seguintes apresentam os valores. Neste caso, todos os dados estão separados pelo caractere `;` .

> **Dica!** Se o arquivo que você baixar vier com acentuação nos nomes das colunas (como acentos ou cedilhas), vale a pena investir um tempinho corrigindo isso. Remover acentos e `ç` facilita o trabalho com a maioria das linguagens de programação, que funcionam melhor com nomes simples e sem caracteres especiais.

É uma tabela cheia de informações, perfeita para começarmos a trabalhar. E o melhor: está exatamente no formato que o nosso importador espera para funcionar. Basta você usar a função do nosso importador e você terá a variável `resultado` do nosso importador com todos os registros deste banco de dados que acabamos de baixar. Vamos visualizar esses dados no console:

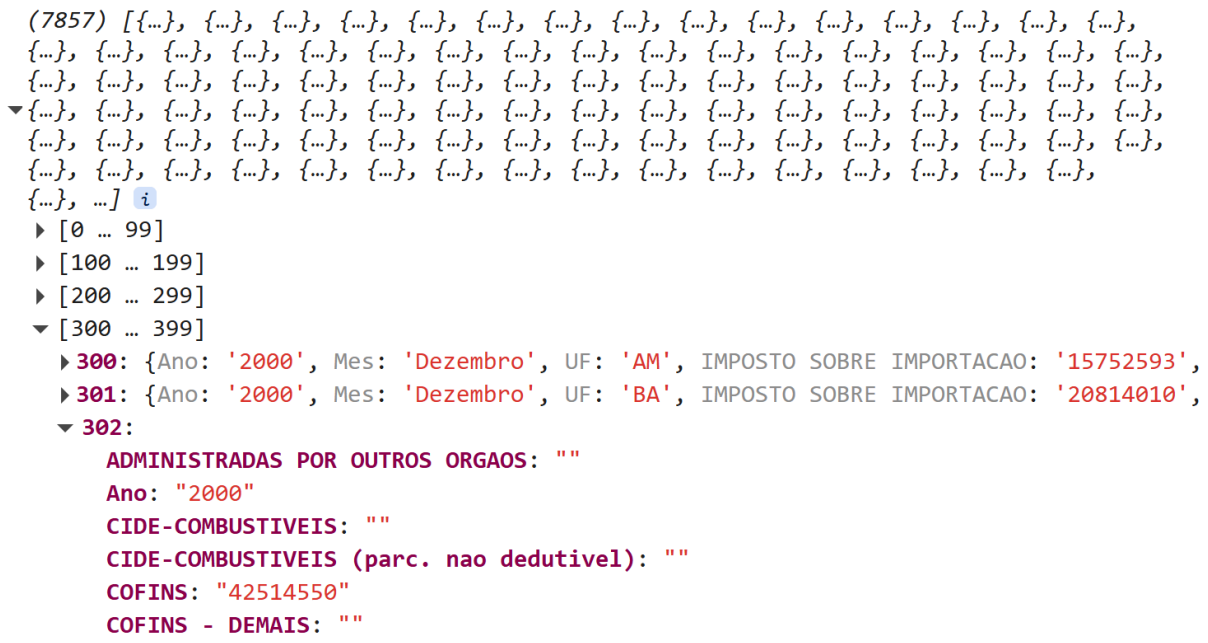


Figura 11.5: Dados por estado no console

Veja a quantidade de registros! São mais de 7 mil entradas, cada uma contendo informações de mais de 15 impostos. E tudo isso organizado em um array de objetos, com dados como o ano, mês, estado e os respectivos valores arrecadados. Isso sim é uma base rica em informações. O tipo de conteúdo que merece um bom Dashboard!

Parece que estamos com a faca e queijo na mão: de um lado o mapa do Brasil já preparado; do outro, os dados de arrecadação estão organizados em um array pronto para uso. Os Dashboards que vamos gerar são sugestivos. Fique à vontade para adaptá-los conforme achar mais interessante para o seu projeto.

11.4 Agregando valores por estado e por imposto

Conhecendo o conteúdo da base de dados

A planilha que estamos utilizando como base de dados compreende valores monetários de arrecadação de impostos no Brasil no período de janeiro de 2000 a março de 2024.

Se você analisar esse arquivo com atenção, perceberá que alguns impostos surgem apenas em determinados períodos; outros apresentam valores zerados em certos anos. Essas e outras mudanças são comuns em bases de dados extensas e com longos intervalos de tempo.

Afinal, ao longo de 24 anos, tivemos mudanças nas leis, surgimento e extinção de tributos, diferentes governos e até possíveis alterações técnicas, como troca de sistemas de gestão, que podem afetar o formato ou a estrutura dos dados. Tudo isso influencia diretamente na qualidade e consistência da base. Por isso, aqui vai mais um reforço importante: **sempre verifique a integridade dos dados antes de gerar qualquer Dashboard.**

Nosso primeiro Dashboard com o mapa

Nosso ponto de partida será um Dashboard interessante e simples, usando os dados que já temos: um mapa com o total arrecadado por estado, ao longo de todo o período, separado por tipo de imposto. A ideia é lapidar esses dados, criando um novo array com a soma dos valores por estado e colorir o mapa de acordo com esses totais.

Vamos fazer isso juntos? Você pode começar um novo código ou aproveitar o mapa que criamos anteriormente. A seguir, apresento os elementos essenciais que devem estar presentes no seu HTML:

```
<!-- Chamadas para o mapa dentro do head -->
<link rel="stylesheet" href="https://unpkg.com/leaflet/dist/leaflet.css"
/>
<script src="https://unpkg.com/leaflet/dist/leaflet.js"></script>

<!--Um campo para fazer o usuário colocar o arquivo .csv-->
<!--Com a chamada para a função que já vai transformar os dados em um
array de objetos chamado resultado-->
<input type="file" id="inputArquivo" class="form-control" accept=".csv"
onchange="lerArquivoPontoEVirgula()">

<!--O botão que vamos trabalhar para gerar o Dashboard-->
<a type="button"  class="btn btn-success" accept=".csv"
onclick="CarregarTotaisPorImposto()">Carregar Totais</a>

<!--Uma div no seu html para o mapa-->
<div id="map" style="width: 100%; height: 350px;"></div>

<!--Um parágrafo para colocar legendas (opcional)-->
<p id="legendas"></p>

<!-- Script do nosso programa -->
<script src="scripts/08_dashboard.js"></script>
```

Esses são os itens para montar seu HTML. Os demais elementos, como títulos, textos explicativos ou ajustes visuais você pode criar à vontade.

> **Atenção!** Meu arquivo JS ficou com o nome `08_dashboard.js` , pois estou seguindo a ordenação didática do código disponível para você no link do repositório deste livro: https://github.com/luctauil/livro_dashboards_com_js. Você pode seguir o seu próprio padrão de nomenclatura para os arquivos.

Vamos declarar algumas variáveis fora de qualquer função, para que fiquem acessíveis em qualquer parte do programa:

```
const map = L.map('map').setView([-14.2350, -51.9253], 3);
var camadaEstados = null;
const somasPorEstado = {};
var max = 0;
var min = 0;
```

Explicando cada uma delas:

- `const map` : inicializa o mapa, já posicionado com coordenadas centralizadas no Brasil;
- `var camadaEstados` : vamos guardar a camada de estados nessa variável. Isso nos permite remover ou atualizar o mapa com facilidade, sem sobrepor camadas ao recarregar os dados;
- `const somasPorEstado` : será o objeto em que iremos acumular as somas dos impostos por estado a partir da leitura do arquivo CSV;
- `var max` e `var min` : serão usados para armazenar o maior e o menor valor de arrecadação. Com esses extremos, conseguimos gerar uma escala de cores proporcional para pintar os estados com destaque visual.

Com base nos valores máximo e mínimo, vamos criar uma função para definir automaticamente a cor conforme o valor total de cada estado. Como estamos lidando com valores financeiros de impostos, faz sentido criar uma escala de cores que vá do verde (valores mais baixos) até o vermelho (valores mais altos). E é isso que faremos com a função `getColor` .

```
function getColor(valor) {
    // Calcula o percentual do valor em relação ao intervalo entre o menor
e o maior
    const percentual = (valor - min) / (max - min);

    // Aumenta o vermelho conforme o valor aumenta
    const red = Math.round(255 * percentual);

    // Diminui o verde conforme o valor aumenta
    const green = Math.round(255 * (1 - percentual));
```

```
    // Retorna a cor no formato RGB: do verde (baixo) ao vermelho (alto)
    return `rgb(${red},${green},0)`;
}
```

Observe que estamos utilizando a notação `rgb()`, que representa cores misturando três componentes: vermelho (*red*), verde (*green*) e azul (*blue*). Confira exemplos simples:

- A cor vermelha pura é `rgb(255, 0, 0)`: máxima intensidade de vermelho e nenhuma de verde ou azul;
- A cor verde pura é `rgb(0, 255, 0)`: máxima intensidade de verde e nenhuma de vermelho ou azul.

A função `getColor` não é só código: tem um pouco de matemática envolvida. Ela normaliza o valor de entrada em um intervalo de 0 a 1 com a fórmula:

```
const percentual = (valor - min) / (max - min);
```

Assim, conseguimos calcular, com exatidão, quanto de vermelho e verde colocar para representar visualmente o valor arrecadado de cada estado. Ou seja: uma função simples, poderosa, e que transforma números em visualização direta.

Vamos montar a função `CarregarTotaisPorImposto()`, que vai pintar nosso mapa com base no imposto escolhido. A ideia é somar os valores por estado, calcular os extremos (mínimo e máximo) e, com isso, gerar a coloração proporcional. Essa função vai:

- Limpar o mapa, caso já exista alguma camada;
- Escolher o imposto que será analisado;
- Somar valores por estado;
- Calcular o maior e menor valor;
- Pintar o mapa com as cores proporcionais;
- Exibir os valores em *tooltips* interativos (aqueles balões que aparecem quando passamos o mouse por cima de algo).

```
function CarregarTotaisPorImposto() {
    if (camadaEstados) { //Esse if verifica se tem algo no mapa
        map.removeLayer(camadaEstados); // Remove a camada do mapa, se
existir
    }

    //Coloque nesta variável o imposto que deseja analisar no Dashboard.
    const impostoSelecionado = 'IMPOSTO TERRITORIAL RURAL';

    // Percorre o array de dados "resultado"
    resultado.forEach(linha => {
        const estado = linha.UF; // Pega o estado atual

        // Garante que o estado esteja inicializado
        if (!somasPorEstado[estado]) somasPorEstado[estado] = 0;

        // Verifica se há valor para o imposto escolhido nesta linha
        if (linha[impostoSelecionado]) {
            // Converte a string com vírgula para número decimal
            let valor = parseFloat(linha[impostoSelecionado].replace(',',
'.'));

            // Soma o valor se for um número válido
            if (!isNaN(valor)) somasPorEstado[estado] += valor;
        }
    });

    // Extrai os valores somados e encontra o máximo e o mínimo para usar
no gradiente de cor
    const valores = Object.values(somasPorEstado);
    max = Math.max(...valores);
    min = Math.min(...valores);

    //...Continuação da função
}
```

Observe que está descrito nos comentários o que cada linha faz. Na linha `const impostoSelecionado = 'IMPOSTO TERRITORIAL RURAL';`, você

pode colocar o nome de qualquer outro imposto disponível nas colunas do arquivo `.csv` para analisá-lo também.

Outros exemplos de impostos visualmente legais para testar são:

- IRRF;
- IPI - FUMO;
- IPI - BEBIDAS;
- ou qualquer outros dos 20 impostos do arquivo.

Nesta primeira etapa da função, já temos no array `somaPorEstado` todos os valores somados de todo o período separado por estado. Agora, na segunda etapa, é hora de desenhar o mapa:

```
function CarregarTotaisPorImposto() {
    //... Etapa anterior da função

    // Busca o arquivo GeoJSON com os contornos dos estados
    fetch('../../dados/br_states.json')
        .then(response => response.json())
        .then(data => {
            // Cria a nova camada no mapa com os dados e estilos definidos
            camadaEstados = L.geoJSON(data, {

                // Define o estilo de cada estado baseado no valor
arrecadado
                style: feature => {
                    const estado = feature.properties.SIGLA;
                    const valor = somasPorEstado[estado] || 0;
                    return {
                        fillColor: getColor(valor), // Cor calculada
dinamicamente
                        weight: 2, // Espessura da borda
                        opacity: 1, // Opacidade da borda
                        color: 'black', // Cor da borda
                        fillOpacity: 0.7 // Transparência da área interna
                    };
                },
```

```
                //todo o bloco onEachFeature é opcional. Esse trecho cria
um balão ao passar o mouse por cima do estado
                onEachFeature: (feature, layer) => {
                    const estado = feature.properties.SIGLA;
                    const valor = somasPorEstado[estado] || 0;

                    layer.bindTooltip(
                        `${estado}: R$ ${valor.toLocaleString('pt-BR', {
minimumFractionDigits: 2, maximumFractionDigits: 2 })}`,
                        { permanent: false, direction: 'center' }
                    );
                }
            }).addTo(map); // Adiciona a camada ao mapa
        });
}
```

E pronto! Com isso, seu mapa será carregado com cores que representam o total arrecadado por estado e, ao passar o mouse (ou tocar no celular), o valor aparece para o usuário.

Esse é o tipo de visualização simples e poderosa que um bom Dashboard deve apresentar.

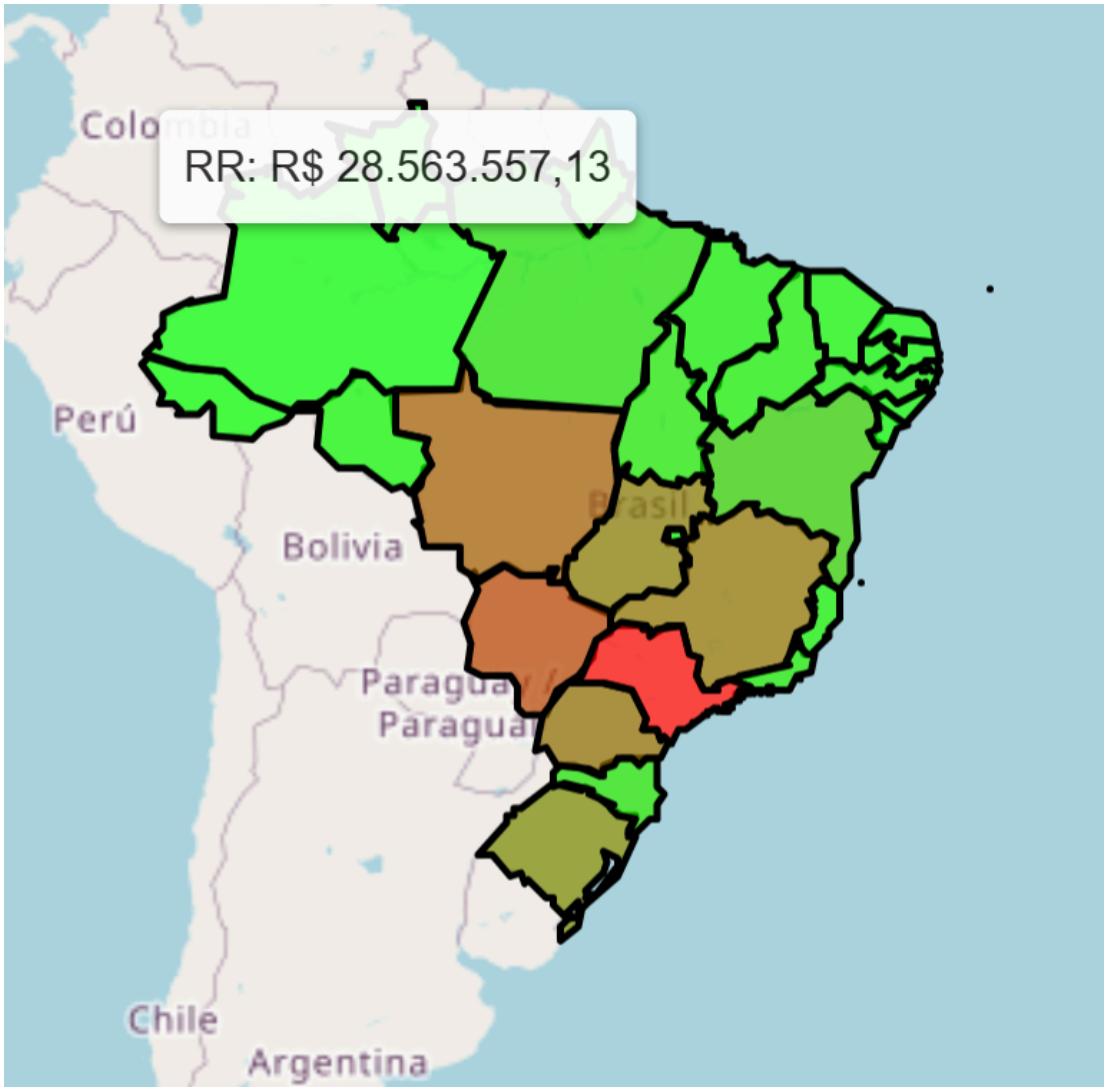


Figura 11.6: Total por imposto de "IMPOSTO TERRITORIAL RURAL"

Adicionando Legenda

Já temos bastante informação no mapa, mas que tal incrementar ainda mais? Embora cada valor seja exibido quando passamos o mouse nos estados, vamos colocar uma legenda?

Faremos isso com uma nova função, para manter o código organizado. Cada funcionalidade com seu objetivo.

```
function Legendas(somas) {
    // Transforma o objeto somasPorEstado em um array de pares [estado,
```

```
valor]
    const listaOrdenada = Object.entries(somas)
        .sort((a, b) => b[1] - a[1]); // Ordena os estados pelo valor
arrecadado, do maior para o menor

    let html = ''; //Variável para montar o HTML

    listaOrdenada.forEach(([estado, valor]) => { //Monta um HTML
        const cor = getColor(valor); //Usa a mesma cor do mapa
        html += `<span style="display: inline-block; width: 12px; height:
12px; border-radius: 50%; background-color: ${cor}; margin-right: 6px;">
</span>`; //Cria um span com a bolinha da cor do estado
        html += `${estado}: R$ ${valor.toLocaleString('pt-BR', {
minimumFractionDigits: 2, maximumFractionDigits: 2 })}<br/>`; //Coloca o
valor na frente da bolinha
    });

    //Insere o conteúdo na tag <p> que deixamos como opcional na tela.
    document.getElementById('legendas').innerHTML = html;
}
```

Para que essa função seja executada junto do carregamento dos dados, basta chamá-la ao final da função `CarregarTotaisPorImposto()`. Chame-a assim: `Legendas(somasPorEstado)` com o parâmetro dos valores somados. Você verá o seguinte resultado:

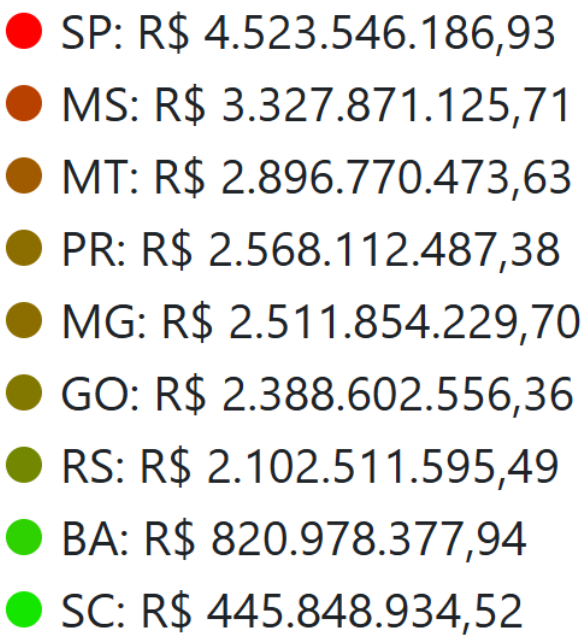


Figura 11.7: Legenda do imposto "IMPOSTO TERRITORIAL RURAL"

Lembra que dissemos que um Dashboard pode contar história?

Esse projeto deu mais trabalho que um simples Dashboard de barras. Em um Dashboard de barras, também conseguiríamos identificar o estado que mais arrecadou com menos código — sim, é verdade — mas perceba a narrativa ao usar o mapa.

Além da informação monetária, é possível ter uma ideia da extensão territorial. É possível observar que estados muito maiores arrecadam muito menos do que estados menores. Às vezes, um estado sozinho arrecada mais do que vários estados juntos. Interessante, não é?

11.5 Criando Dashboards Temporais

Se o nosso primeiro *insight* mostrou quem arrecada mais, que tal agora conferir como essa arrecadação evolui ao longo dos anos? A

ideia é analisar o mesmo imposto, porém com recorte ano a ano. Assim conseguimos responder perguntas como:

- Qual estado liderava a arrecadação em 2015?
- O estado que mais arrecadou em 2023 sempre esteve no topo?
- Algum estado despencou ou teve crescimento rápido nos últimos anos?

Essas perguntas são comuns no dia a dia de analistas e tomadores de decisão. E agora você vai aprender a respondê-las com um simples mapa interativo.

Para começar, precisamos incluir um seletor de ano na nossa tela. Assim, o usuário pode escolher qual ano deseja visualizar no mapa. Vamos adicionar esse código ao HTML, logo acima de um novo botão de ação:

```
<select id="selectAno" class="form-control">
    <!-- Adicione aqui os anos disponíveis na sua base -->
    <option value="2020">2020</option>
    <option value="2021">2021</option>
    <option value="2022">2022</option>
    <option value="2023">2023</option>
</select>

<a class="btn btn-primary" onclick="CarregarTotaisPorAno()">Ver
Evolução</a>
```

Agora que temos a interface, vamos ao código da função `CarregarTotaisPorAno()`.

A lógica dessa função é parecida com a função `CarregarTotaisPorImposto()` que fizemos anteriormente. Vamos filtrar os dados por ano, somar os valores apenas daquele ano e pintar o mapa com essas informações.

```
function CarregarTotaisPorAno() {
    if (camadaEstados) map.removeLayer(camadaEstados);
```

```
    const anoSelecionado = document.getElementById('selectAno').value;
    const impostoSelecionado = 'IMPOSTO TERRITORIAL RURAL';

    const somasPorEstadoAno = {};

    resultado.forEach(linha => {
        if (linha.Ano === anoSelecionado) {
            const estado = linha.UF;
            let valor = parseFloat(linha[impostoSelecionado]?.replace(',',
'.') || 0);

            if (!somasPorEstadoAno[estado]) somasPorEstadoAno[estado] = 0;
            if (!isNaN(valor)) somasPorEstadoAno[estado] += valor;
        }
    });

    console.log(`Soma por ano`, somasPorEstadoAno);

    const valores = Object.values(somasPorEstadoAno);
    max = Math.max(...valores);
    min = Math.min(...valores);

    fetch('../../dados/br_states.json')
        .then(r => r.json())
        .then(data => {
            camadaEstados = L.geoJSON(data, {
                style: feature => {
                    const estado = feature.properties.SIGLA;
                    const valor = somasPorEstadoAno[estado] || 0;
                    return {
                        fillColor: getColor(valor),
                        weight: 2,
                        opacity: 1,
                        color: 'black',
                        fillOpacity: 0.7
                    };
                },
                onEachFeature: (feature, layer) => {
                    const estado = feature.properties.SIGLA;
```

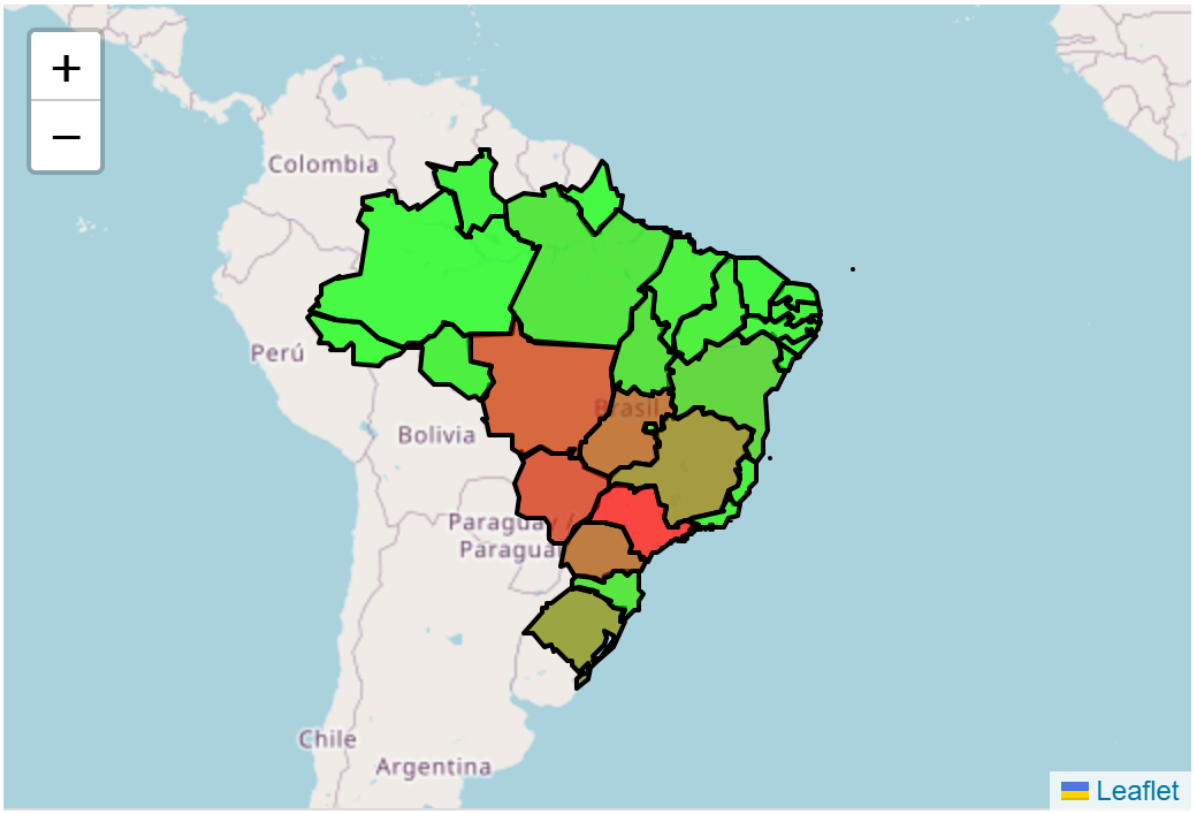
```
                    const valor = somasPorEstadoAno[estado] || 0;
                    layer.bindTooltip(
                        `${estado} (${anoSelecionado}): R$
${valor.toLocaleString('pt-BR', {minimumFractionDigits: 2})}`
                    );
                }
            }).addTo(map);
        });

    Legendas(somasPorEstadoAno);
}
```

Veja que, desta vez, ao final da função, chamei novamente a função `Legendas(somasPorEstadoAno)`, agora passando os novos valores somados. Assim, a legenda será adicionada automaticamente.

Conforme construímos o código de forma mais organizada, conseguimos aproveitar melhor suas partes, tornando o desenvolvimento mais eficiente.

Você verá o mapa assim:



- SP: R$ 472.763.876,61
- MS: R$ 401.103.467,45
- MT: R$ 380.356.594,00
- GO: R$ 328.963.787,35

Figura 11.8: Análise do imposto "IMPOSTO TERRITORIAL RURAL" por ano

Esse novo tipo de visualização transforma nosso mapa em um Dashboard temporal. A cada ano selecionado, uma nova história pode surgir:

- Pode indicar que um estado teve um aumento expressivo na arrecadação em determinado ano;
- Pode revelar quedas bruscas que exigem atenção ou investigação;
- Pode mostrar que certos estados mantêm a liderança, enquanto outros não param de crescer.

Tudo isso, com apenas um clique no seletor de ano.

Você pode, inclusive, experimentar colocar dois botões lado a lado: um para Totais por Imposto e outro para Evolução por Ano, permitindo que o usuário escolha que tipo de *insight* deseja observar. As opções são inúmeras.

11.6 Personalizando interatividade e estilos do mapa

Os Dashboards que criamos até aqui focaram em dados financeiros. Conseguimos visualizar, por exemplo, quais estados arrecadam mais impostos, como isso mudou ao longo do tempo e até como os dados ganham força quando combinados com a visualização geográfica.

Esses *insights* contam algumas histórias que podem despertar ainda mais a curiosidade. Por exemplo: percebemos que alguns estados pequenos arrecadam mais do que estados maiores. Será que isso significa que há espaço para crescimento nesses outros estados? Quais fatores contribuem para esse cenário? Existe relação ambiental ou com o tamanho da população? E se quisermos ir além do dinheiro?

A boa notícia é que o próprio mapa pode nos ajudar com isso.

Até aqui, usamos o arquivo GeoJSON apenas como uma base visual para desenhar o contorno dos estados. Mas esse arquivo pode conter muito mais do que coordenadas geográficas. No nosso caso, o próprio GeoJSON já traz algumas informações demográficas básicas, como o número de homens e mulheres por estado. E, com um pouco de criatividade, podemos gerar novos Dashboards sem nem precisar carregar um CSV externo.

Nosso objetivo nesta nova seção será extrair dados do próprio GeoJSON e montar visualizações que tragam outras dimensões de

informação: proporção populacional, distribuições por sexo e até padrões curiosos sobre a demografia brasileira que muitas vezes passam despercebidos. Vamos começar?

> No nosso repositório de códigos, esse novo mapa está separado na página do HTML `09_dashboard_mapa_demografico.html` dentro da pasta `dashboards`. Lá os códigos estão organizados de forma granular para facilitar o aprendizado com informações individualizadas. Fique à vontade para juntar os mapas e mesclar as análises.

Para este Dashboard, os elementos de HTML que vamos precisar são parecidos com os do anterior:

```
<!-- Chamadas para o mapa dentro do head -->
<link rel="stylesheet" href="https://unpkg.com/leaflet/dist/leaflet.css"
/>
<script src="https://unpkg.com/leaflet/dist/leaflet.js"></script>

<!-- Botões para interagir com o mapa -->
<a onclick="DadosPopulacionais()">Dados Populacionais - Totais</a>
<a onclick="DadosPopulacionais('Homens')">Dados Populacionais -
Masculino</a>
<a onclick="DadosPopulacionais('Mulheres')">Dados Populacionais -
Feminino</a>

<!-- div onde o mapa será renderizado -->
<div id="map" style="width: 100%; height: 350px;"></div>

<!-- Parágrafo para exibir as legendas -->
<p id="legendas"></p>

<!-- Link para o arquivo JS com a lógica do Dashboard -->
<script src="scripts/09_dashboard.js"></script>
```

Neste mapa, como o carregamento populacional é bem parecido entre homens, mulheres e o total, faremos os três mapas a partir da

mesma função, passando apenas o tipo desejado como parâmetro. As configurações iniciais continuam iguais às anteriores: posicionamento central no Brasil, variáveis `max` e `min` para controle de cores e `camadaEstados` para atualizar o mapa dinamicamente.

```
const map = L.map('map').setView([-14.2350, -51.9253], 3); //Posição
inicial sobre o Brasil

var camadaEstados = null;
var max = 0;
var min = 0;

/* Carrega o Mapa */
onload = () => {
    L.tileLayer('https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png', {
        maxZoom: 20,
    }).addTo(map);
}
```

Já na função que gera as cores, vamos trocar a paleta verde/vermelha usada para dados financeiros por uma escala que transmite densidade populacional, indo do amarelo claro ao vermelho escuro/marrom, como normalmente vemos em mapas de calor demográficos.

```
function getColor(valor) {
    if (!valor || isNaN(valor) || valor === 0) {
        return 'rgb(230,230,230)'; // cinza claro para ausência de dados
    }

    const intervalo = max - min || 1;
    const percentual = (valor - min) / intervalo;

    // Escala: de amarelo claro a vermelho/marrom escuro
    const r = Math.round(255); // vermelho no máximo
    const g = Math.round(255 * (1 - percentual)); // reduz o verde
conforme o valor aumenta
    const b = Math.round(100 * (1 - percentual)); // azul discreto para
escurecer o tom
```

```
    return `rgb(${r},${g},${b})`;
}
```

Na função de legendas, fazemos apenas uma pequena adaptação: não usamos R$, pois agora lidamos com quantidade de habitantes.

```
function Legendas(somas) {
    const listaOrdenada = Object.entries(somas)
        .sort((a, b) => b[1] - a[1]); // do estado mais populoso ao menos

    let html = '';

    listaOrdenada.forEach(([estado, valor]) => {
        const cor = getColor(valor);
        html += `<span style="display: inline-block; width: 12px; height:
12px; border-radius: 50%; background-color: ${cor}; margin-right: 6px;">
</span>`;
        html += `${estado}: ${valor.toLocaleString('pt-BR')}
habitantes<br/>`;
    });

    document.getElementById('legendas').innerHTML = html;
}
```

Agora, vamos à função responsável por carregar os dados do próprio GeoJSON, calcular os valores, gerar os estilos no mapa e exibir os *tooltips* (balões de informação) personalizados de acordo com o tipo (Total, Homens, Mulheres):

```
function DadosPopulacionais(tipo = 'Total') {
    if (camadaEstados) map.removeLayer(camadaEstados); // Remove a camada
do mapa anterior, se existir

    const somasPopulacao = {};

    fetch('../../dados/br_states.json') //Caminho para o arquivo GeoJSON
        .then(response => response.json())
        .then(data => {
```

```
            console.log(data);

// Calcula a população por estado, de acordo com o tipo selecionado
            data.features.forEach(feature => {
                const estado = feature.properties.SIGLA;
                const valor = parseInt(feature.properties[tipo] || 0);
                somasPopulacao[estado] = valor;
            });

// Atualiza os valores mínimos e máximos para controle da escala de cores
            const valores = Object.values(somasPopulacao);
            max = Math.max(...valores);
            min = Math.min(...valores);


// Cria a camada do mapa com estilos personalizados
            camadaEstados = L.geoJSON(data, {
                style: feature => {
                    const estado = feature.properties.SIGLA;
                    const valor = somasPopulacao[estado] || 0;

                    return {
                        fillColor: getColor(valor),
                        weight: 2,
                        opacity: 1,
                        color: 'black',
                        fillOpacity: 0.7
                    };
                },
                onEachFeature: (feature, layer) => {
                    const estado = feature.properties.SIGLA;
                    const valor = somasPopulacao[estado] || 0;

                    const labelTipo = tipo === 'Total' ? 'habitantes'
                                    : tipo === 'Homens' ? 'homens'
                                    : tipo === 'Mulheres' ? 'mulheres'
                                    : tipo.toLowerCase();

                    layer.bindTooltip(
```

```
                        `${estado}: ${valor.toLocaleString('pt-BR')}
${labelTipo}`,
                        { permanent: false, direction: 'center' }
                    );
                }
            }).addTo(map);

            Legendas(somasPopulacao);
        });
}
```

O resultado fica legal:

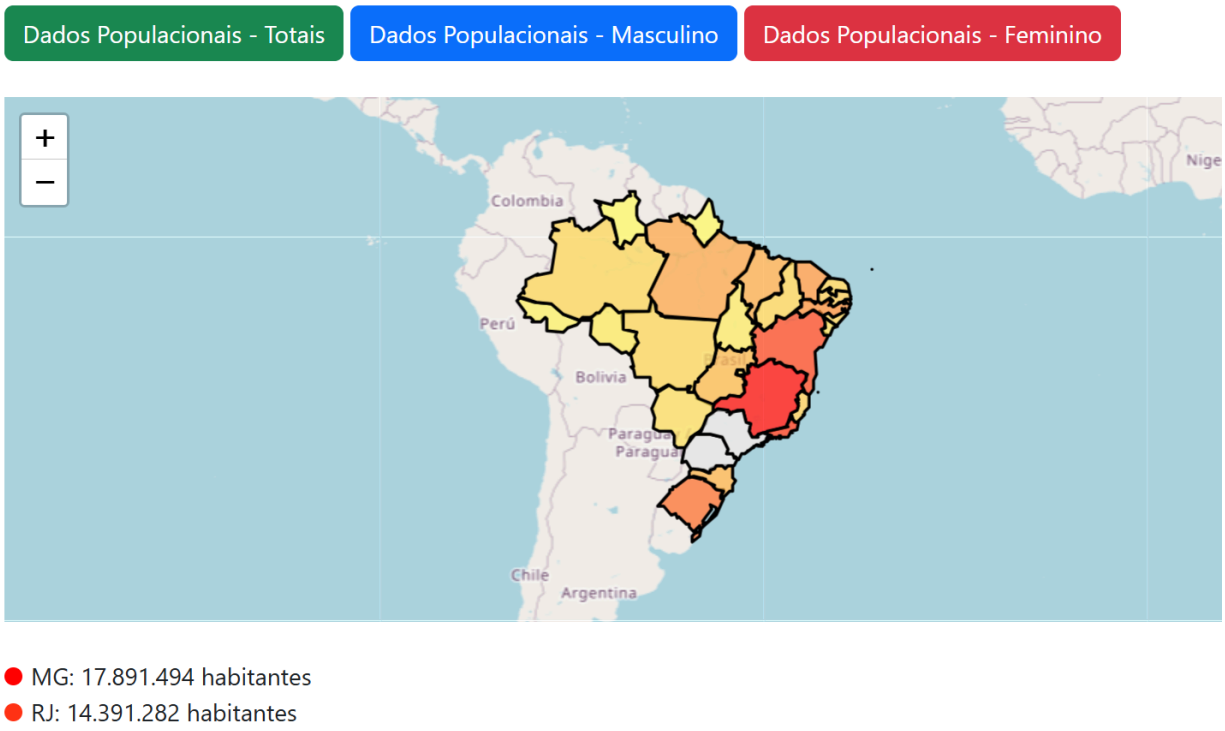


Figura 11.9: Dados demográficos do Brasil

💡 **Observação:** repare que nos estados de São Paulo e Paraná neste mapa, ambos aparecem em cinza claro. Isso acontece porque, no nosso arquivo GeoJSON, os dados demográficos desses estados estão ausentes. Essa situação reforça um ponto importante no desenvolvimento de Dashboards: nem sempre os dados vêm completos ou perfeitos, e nosso código precisa estar preparado para lidar com essas lacunas.

Se a função `getColor()` não estivesse programada para tratar casos em que o valor é `0`, `null` ou `undefined`, o mapa poderia apresentar erros graves ou nem mesmo ser carregado corretamente. Esse tipo de precaução — pensar nas exceções, prever ausências e oferecer alternativas visuais claras — é o que diferencia um código funcional de um código robusto e confiável.

Fica aqui a dica: trate os dados com carinho e desconfiança ao mesmo tempo. Planeje para o melhor cenário, mas esteja sempre preparado para o pior.

E agora que você domina as ferramentas para isso, que tal pensar nas suas próprias perguntas investigativas?

- Será que existe alguma relação entre a população de um estado e sua arrecadação?
- Os estados com menor densidade populacional são também os que menos arrecadam?
- Como ficaria um mapa com a arrecadação per capita por estado?

Este é o tipo de cruzamento que começa a surgir naturalmente quando você visualiza os dados de diferentes formas. E a melhor parte: você já tem o conhecimento necessário para começar a testar essas ideias.

Agora é com você: explore, combine, questione — e visualize.

11.7 Conclusão do projeto: Dashboard Geográfico

Chegamos ao fim da jornada.

Neste livro, você percorreu um caminho completo: começou entendendo os conceitos fundamentais da web, explorou as estruturas básicas de HTML, CSS e JavaScript, aprendeu a organizar seus arquivos, escrever funções, trabalhar com dados externos e, enfim, gerar Dashboards interativos com informações reais.

Você viu que não é preciso dominar dezenas de ferramentas complexas ou pagas para criar visualizações impactantes. Apenas com o navegador, um editor de código e um pouco de paciência já é possível transformar dados em narrativas visuais poderosas.

Mais do que ensinar a fazer Dashboards, este livro mostrou como pensar um projeto de visualização: como estruturar, buscar os dados, desenvolver um raciocínio visual e utilizar o código como uma ponte entre a informação e a interpretação.

Além disso, todas as ferramentas usadas aqui são gratuitas, acessíveis e leves. Nada de frameworks complicados ou dependências desnecessárias. O foco foi proporcionar mais autonomia. E a ideia central sempre foi colocar o poder da visualização de dados nas suas mãos.

> Porque não basta ver os dados, é preciso entendê-los.

A fixação do conteúdo vem com a prática. Volte aos capítulos, recrie os exemplos, experimente modificá-los, trocar os dados, testar outros formatos. Cada tentativa é um passo rumo à fluência.

Espero sinceramente que este material tenha contribuído com seu crescimento, não apenas como desenvolvedor ou designer de

Dashboards, mas como alguém mais curioso e capaz de contar histórias com dados.

Obrigado por confiar neste conteúdo.

Nos encontramos em seus próximos projetos!